# High Performance XML Parsing Using Parallel Bit Stream Technology

Robert D. Cameron Kenneth S. Herdy Dan Lin

School of Computing Science, Simon Fraser University

# Abstract

Parabix (parallel bit streams for XML) is an open-source XML parser that employs the SIMD (single-instruction multiple-data) capabilities of modern-day commodity processors to deliver dramatic performance improvements over traditional byte-at-a-time parsing technology. Byte-oriented character data is first transformed to a set of 8 parallel bit streams, each stream comprising one bit per character code unit. Character validation, transcoding and lexical item stream formation are all then carried out in parallel using bitwise logic and shifting operations. Byte-at-a-time scanning loops in the parser are replaced by bit scan loops that can advance by as many as 64 positions with a single instruction.

A performance study comparing Parabix with the open-source Expat and Xerces parsers is carried out using the PAPI toolkit. Total CPU cycle counts, level 2 data cache misses and branch mispredictions are measured and compared for each parser. The performance of Parabix is further studied with a breakdown of the cycle counts across the core components of

the parser. Prospects for further performance improvements are also outlined, with a particular emphasis on leveraging the intraregister parallelism of SIMD processing to enable intrachip parallelism on multicore architectures.

# 1 Introduction

Parallel bit stream technology offers a promising way to break out of the performance bottleneck associated with traditional byte-at-a-time text processing on today's commodity processors. On these processors, SIMD registers are commonly 128 bits wide, yet traditional text processing works with only 8 bits at a time. Beyond this, the information computed per byte can often be as low as a single bit. For example, in scanning for a particular character, such as the opening angle bracket of an XML markup item, each loop iteration effectively gives us only one bit of information: whether we have found the character in question or not. Thus, there is often a dramatic mismatch between available processor resources and the amount of information that is computed in byte-at-atime text processing.

Now consider putting all 128 bits of a SIMD register to work in our character scanning application. That is, suppose that we simultaneously calculate whether or not each of the next 128 bytes represents an opening angle bracket,

Copyright © 2008 Robert D. Cameron, Kenneth S. Herdy and Dan Lin. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made. This is a prepublication version of a paper to appear in the Proceedings of CASCON 2008

storing the result as a sequence of 128 onebit results in the register. Extending this concept beyond the length of a single 128-position block, we define a parallel bit stream as a sequence of bit values in one-to-one correspondence with the character code units of our XML data stream. We can then replace the byte-ata-time scanning loop with bit scan operations on the bit stream. In fact, most modern processors provide such an operation as a builtin instruction operating on 32 or 64 bits at a time. Thus, given an opening angle bracket bit stream, it can be possible to identify the position of the next opening angle bracket anywhere within the next 32 to 64 positions in a single operation.

As our measurements show, the amortized cost of producing a basis set of parallel bit streams from which the opening angle bracket and other useful bit streams may be derived is on the order of one CPU cycle per XML input byte on commodity processors. This is a very small overhead compared with the 100 CPU cycles per byte range generally reported as the cost of XML parsers such as libXML2 [15] and Xerces[8]. A complete set of parallel bit streams useful for XML parsing then can be produced within one additional cycle per byte. Within about 0.5 cycles per byte, parallel bit streams can be further be used to validate UTF-8 character representations as well as to identify valid characters that are illegal in XML. As needed, UTF-8 to UTF-16 transcoding can also be implemented in just a few cycles per byte depending on input characteristics [4]; this transcoding operation is frequently cited as a major bottleneck responsible for 30% or more of XML parsing costs[9, 10, 12]. Other applications of parallel bit streams to XML processing include parallel hash value computation and parallel regular expression matching.

A principal focus of our research work is the development of an open-source XML parser as a vehicle to systematically investigate the use of parallel bit stream technology in high-performance XML processing. The ultimate goal is to provide a complete implementation fully conformant with the relevant XML standards, portable to a number of architectures and organized to leverage the intraregister parallelism of SIMD processing to further exploit

the intrachip parallelism of multicore processors. At present, Parabix is nearing completion as a nonvalidating XML 1.0 processor comparable to Expat [5], but with broader support for alternative character sets and better conformance to the XML conformance test suite. An initial implementation of DTD processing and corresponding content validation has also been made. Parabix is also being commercialized using a patentleft strategy through International Characters, Inc., a spin-off company from Simon Fraser University. Parabix is a trademark of International Characters.

The focus of this paper is to describe the basic design of Parabix, focusing on the application of parallel bit stream technology and to study the XML processing performance of Parabix in relation to other XML parsers and also in relation to the ultimate performance that might be achieved as Parabix evolves. Section 2 of this paper discusses the basic architecture of Parabix together with the core techniques used in nonvalidating XML processing. Section 3 then moves on to describe the methodology of our performance study and discuss the high-level results of our head-to-head study versus the Expat and Xerces open-source parsers. Section 4 then considers performance aspects of the individual components of Parabix in detail. Section 5 concludes the paper with a summary of results and directions for further work.

# 2 Parabix Structure

Figure 1 shows the overall architecture of the parabix parser. The XML Interface Module controls the interpretation and processing of an XML document by first extracting XML model information (the XML declaration and Document Type Definition) and supplying this information to the XML Model Processor. The Model Processor interprets this information and stores information with respect to element, attribute, entity and notation names in the Symbol Table Module. The processing of the document content is then initiated by supplying this content as a byte stream to the Parallel Bit Stream Module. For simplicity, this may be considered to be a UTF-8 byte stream, al-

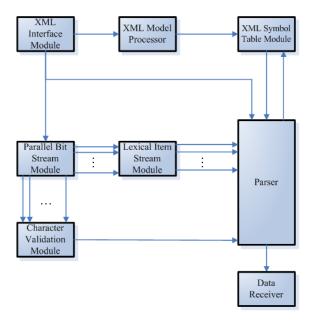


Figure 1: Parabix Architecture

though parabix has a character set architecture (not illustrated) that permits efficient parsing with a wide variety of ASCII or EBCDIC based character sets.

The Parallel Bit Stream Module generates a basis set of eight parallel bit streams that together represent the document content. Bit stream 0 represents bit 0 of each byte of the document, bit stream 1 represents bit 1 of each byte and so on. These streams are then applied as input to both the Lexical Item Stream Module and the Character Validation Module. The Lexical Item Stream Module computes streams useful for identifying lexical items in the parsing process, that is, streams to identify the start of markup when scanning text, the following character after a name when scanning a name, the closing quote when scanning an attribute value, hyphen and question mark streams for scanning comments and processing instructions and a CDATA delimiter stream for scanning both the content of CDATA sections and validating the content of text generally. The Character Validation Module separately uses the parallel bit streams to validate that all UTF-8 byte sequences are well-formed and that the encoded characters are legal in XML.

The Parser uses the lexical item streams in conjunction with the original byte stream to parse the document. Typically the parser alternates between bit scan operations on a lexical item stream followed by byte-based tests on the UTF-8 byte stream. For example, in processing text, the parser scans for an occurrence of a MarkupStart character ("<" or "&"). Given the exact position of the markup start, tests on the input byte stream identify the particular markup item as a start tag, end tag, processing instruction, comment, CDATA section or reference. If a start tag is identified, input is advanced by one and a scan is then made to the next NameFollow character to determine the length of the name. Byte tests on the characters following the name can then distinguish between normal start tags without attributes, empty element tags without attributes or tags that have nonempty attribute lists. The overall structure of the scanner is quite similar to a recursive descent style parser in which byte-ata-time scan loops are all replaced by bit scans of lexical item streams.

During parsing, the Parser identifies all names of entities, elements, attributes and notations and consults the symbol table with respect to their prior definition. From the standpoint of a nonvalidating XML processor, the primary purpose of symbol table lookup is to provide fast validation that names have the correct XML name syntax if they have been encountered previously in the document. This avoids byte-by-byte checking of names for correct XML name syntax except for the first time the name is encountered in the document (or during model processing). As the parser proceeds to identify markup items, markup action routines are called to provide the parsed data to a Data Receiver module which implements the application. In essence, these markup action routines implement a SAX-style event-based

Parabix processes XML content using a buffered, streaming model. Buffers are sized so that a buffer full of XML byte data together with the derived parallel bit streams and lexical item streams fit well within the L2 data cache. In the current implementation, an input buffer of 100K XML data bytes is used in conjunction with a processor having a 2M L2 cache. The parallel bit stream operations are organized in separate stages. The Parallel Bit Stream Mod-

ule first transforms the 100K input byte stream into the eight basis bit streams of 12.5K each. The Character Validation Module is then given control to find any invalid or illegal characters within the buffer. After these two stages complete, the Lexical Item Stream module is given control to produce the lexical stream for the entire buffer before control is passed on to the parser. The following subsections describe various aspects of bit stream processing in more detail.

## 2.1 Transposition

The key to parabix performance is to work with parallel bit streams. Efficient implementation of transposition from serial byte stream to bit streams is crucial to any application using this technique [4].

The transposition uses 24 SIMD operations in the ideal architecture to transpose 128 serial bytes to parallel bit streams in three stages. Each of the stages includes 8 SIMD pack instructions that combine two consecutive registers of data by extracting n/2 bits from each n bit field. In the first stage, packing operations on 8 bit fields are used to generate two nybble streams. One of them comprises the high nybbles from all the bytes in the byte stream while the other comprises the low nybbles. In the second stage, the two nybble streams are further packed into four pair streams by SIMD pack operations on 4 bit fields. Each of the generated streams contains 2 bits from each byte in the original byte stream. The final stage completes the transposition by packing 2 bit-fields so that all the 8 parallel bit streams are formed. Details and algorithm variations are described in the technical report[3].

#### 2.2 Character Validation

In the Character Validation Module, we need to verify the well-formedness of UTF-8 sequences as well as the XML characters. The process includes prefix and suffix matching and ruling out the illegal characters.

Figure 2 shows an simple example of verification for prefix and suffix matching of UTF-8 sequences. UTF-8 is a variable-length encoding form of Unicode in which one to four 8-bit code

unit (bytes) are used to represent each character. In the case of single byte sequence, the byte values are in the range 0x00-0x7F (first bit is 0), so in Figure 2 the third and fifth bytes are single byte sequences. Bytes with first three bits 110 are the prefixes of two byte sequences and bytes with first two bits 10 are the suffixes of any multibyte sequences, which means the first and fourth byte in Figure 2 are prefix of two byte sequences and the second one is a suffix.

Therefore, to calculate the prefix and suffix classifications of a character stream consisting of single byte and two byte sequences, only three transposed parallel bit streams, bit0, bit1 and bit2 are needed.

```
prefix2 = bit0 & bit1 & (! bit2)
suffix = bit0 & (! bit1)
```

All prefixes of two byte sequences should followed with one suffix. Therefore, in Figure 2 the fifth byte should be a suffix, but the fifth bits of the suffix stream shows that the byte at the fifth position is not a suffix, which means there is a mismatch of the prefix and suffix. The mismatch is detected by performing an xor operation on the shifted prefix and the suffix to produce an error mask.

```
error_mask = ( prefix2 >> 1 ) ^ suffix
```

In a similar fashion, an error mask for threeand four-byte sequences may also be determined by matching suffixes against expectations established by the prefixes.

The error mask is then extended by checking for illegel prefixes, suffixes and XML characters. For example, 0xC0 and 0xC1 are invalid prefixes.

```
COC1 = prefix2 & !(bit3|bit4|bit5|bit6)
error_mask = error_mask | COC1
```

#### 2.3 Lexical Item Streams

A key role of parallel bit streams in parabix is to generate lexical item streams that facilitate scanning operations. For example, a LeftAngle lexical item stream may be formed to identify those character or code unit positions at which an "<" character occurs. Calculating LeftAngle stream for 128 bytes includes 7 SIMD operations.

	1102222	IUAA	*****		UA	OAAAAAA		IIUXXXXX	02424242424	•••••
bit0		1	1	0	1	0				
bit1		1	0	x	1	x				
bit2		0	x	x	0	x				
prefi	x2	1	0	0	1	0		_ ←	prefix2 = bit0 & bi	it1 & (! bit2)
suffix	X	0	1	0	0	0			suffix = bit0 & (! t	pit1)
error_	mask	0	0	0	0	1		_ ←	error_mask = ( pre	efix2>>1) ^ suffix

110xxxxx

0xxxxxxx

0xxxxxxx

Figure 2: Prefix and Suffix Matching Validation

```
temp1 = simd_or(bit[0], bit[1]);
                                                 temp5 = simd_or(bit[6], bit[7]);
temp2 = simd_and(bit[2], bit[3]);
                                                 temp6 = simd_andc(temp4, temp5);
temp3 = simd_andc(temp2, temp1);
                                                 LAngle = simd_and(temp3, temp6);
temp4 = simd_and(bit[4], bit[5]);
temp5 = simd_or(bit[6], bit[7]);
                                                 temp8 = simd_and(temp4, temp7);
temp6 = simd_andc(temp4, temp5);
                                                 RAngle = simd_and(temp3, temp8);
LAngle = simd_and(temp3, temp6);
```

Bit scan operations may then be used by the parser on the LeftAngle stream to identify the position of the next "<" character whenever necessary.

110xxxxx

10xxxxxx

Lexical item streams differ from traditional streams of tokens in that they are bitstreams that mark the positions of tokens, whitespace or delimiters. Differentiation between the actual tokens that may occur at a particular point (e.g., the different XML tokens that begin "<") is performed using multicharacter recognizers on the bytestream representation.

Common subexpressions are frequent between various bitstreams. Therefore, the average number of operations taken by each markup items is much less than 7. For example, it takes 10 operations to form both LeftAngle and RightAngle streams.

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_and(bit[2], bit[3]);
temp3 = simd_andc(temp2, temp1);
temp4 = simd_and(bit[4], bit[5]);
```

```
temp7 = simd_andc(bit[6], bit[7]);
```

There are also optimizations that apply for recognition of character ranges. For example, determination of character range streams for control characters (codepoints 0x00-0x1F) and digits (code points 0x30-0x39) requires only 8 operations in total.

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_or(temp1, bit[2]);
Contrl = simd_andc(simd_const_1(1), temp2);
temp3 = simd_and(bit[2], bit[3]);
temp4 = simd_andc(temp3, temp1);
temp5 = simd_or(bit[5], bit[6]);
temp6 = simd_and(bit[4], temp5);
Digit = simd_andc(temp4, temp6);
```

Overall, the full set of lexical item stream computations requires only 67 logical and shift operations per 128 byte block.

# 3 Parser Comparisons

## 3.1 Methodology

This study evaluates the performance of three C/C++ based, event-driven, stream-oriented XML parsers. Parabix is compared against the Xerces version 2.8.0 (SAX2 API) parser and the Expat version 2.0.1 parser. Xerces [2] is a validating open source XML parser written in C++ by the Apache project. Expat [5] is a non-validating XML parser library written in C. Parabix, Xerces and Expat each provide an event-based application programming interfaces in which an application can registers handlers for specific XML parsing events such as element start events, element end events and character data events. Since the Expat parser is non-validating, the DTD validation feature of Xerces was disabled for the purposes of fair comparison.

The application developed and used to evaluate parsing performance in this study is a simple XML document statistics gathering application. This application calculates the count and average length of each logical component of an input XML document instance. The design choice to develop this application is based on the following reasoning. The application serves a useful purpose, processes every XML markup and content component in a source XML instance, and avoids any additional costs due to non-parsing related computation. In addition, to eliminate disk access I/O costs, each XML document instance was stored in a contiguous memory buffer prior to parsing.

#### 3.2 Test Environment

Performance experiments were conducted on a 2.1 GHz Intel Core 2 Duo processor desktop machine with 2GB of available memory. Table 1 and Table 2 describe in detail the hardware and software configuration of the performance evaluation environment respectively. The Performance Application Programming Interface (PAPI) Version 3.5.0 [1] toolkit was installed on the test system to facilitate the collection of hardware performance monitoring statistics. In addition, the Linux x86 Performance Monitoring Counters Driver known as perfetr [11] was installed to facilitate access

to the performance monitoring counters on the Linux x86 platform.

Intel Core 2 Duo pro-				
cessor 6400				
1066				
2128				
Integrated				
4 core, 2 chip, 2				
cores/chip				
1,2 chip				
32  KB I + 32  KB D on				
chip per core				
2048 KB (I+D) on chip				
None				
None				
2 GB DIMM				
SCSI				
None				

Table 1: Hardware Configuration

Operating System	Ubuntu 7.10 (Linux				
	x86)				
Compiler	GCC version 4.1.3				
Auto Parallel	No				
File System	NTFS				
System State	Default				
Base Pointers	64-bit				
Peak Pointers	64-bit				
Other Software	PAPI Version 3.5.0,				
	perfctr				

Table 2: Software Configuration

## 3.3 Hardware Events

The key hardware events evaluated in this performance analysis include the following events.

- Instructions Completed Reported as the number of instructions retired per byte.
- Processor Cycles Reported as the number of processor cycles executed per byte.
- Conditional Branches Reported as the number of conditional branches per byte.

File Name	dewiki.xml	jawiki.xml	roads.gml	po.xml	soap.xml
File Type	document	document	data	data	data
File Size (kB)	66240	7343	11584	76450	2717
Markup Item Count	406792	74882	280724	4634110	18004
Attribute Count	18808	3529	160416	463397	30001
Avg. Attribute Size	8	8	6	5	9
Markup Density	0.07	0.13	0.57	0.76	0.87

Table 3: XML Document Characteristics

- Branch Mispredictions Reported as the number of mispredicted conditional branch instructions per byte.
- Caches Misses Reported as the number of L1 and L2 data cache misses per byte respectively.

For additional information on these hardware events refer to the IA-32 Intel Architecture Optimization Reference Manual [7].

#### 3.4 Test Data Characteristics

XML document instances are traditionally classified as either data-oriented or document-oriented [6]. Data-oriented instances are typically used for the exchange of database records and are intended to be processed by machines. In contrast, document-oriented instances contain information intended for publication. Data-oriented XML are characterized by a higher markup density as well as a larger number of attributes. However, it is noted that in general many XML instances may not fit neatly into one category or the other.

Table 3 shows the document characteristics of the XML instances selected for this performance study. The jawiki.xml and dewiki.xml XML files represent document-oriented XML instances of Wikimedia books, written in German and Japanese, respectively. The remaining files are data-oriented. The roads.gml file is an instance of Geography Markup Language (GML), a modeling language for geographic systems as well as an open interchange format for geographic transactions on the Internet [13]. The po.xml file is an example of purchase order data, while the soap.xml file contains a large SOAP message. Markup density is defined as the ratio of the total markup contained within

an XML file to the total XML document size. This metric is reported for each document.

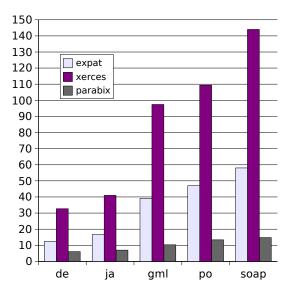


Figure 3: CPU Cycles Per Byte

#### 3.5 Performance Results

Figure 3 shows the result of our overall performance comparison between Expat, Xerces and Parabix on our five test XML files, assessed in terms of CPU cycles per byte of UTF-8 input. In each case Parabix is dramatically faster, ranging from twice as fast as Expat on document-oriented input to an order of magnitude faster than Xerces on data-oriented input. The comparison is not entirely fair to Xerces, as it transcodes to UTF-16, while Expat and Parabix do not. Byte at a time transcoding is quite expensive, which would narrow the

gap between Expat and Xerces. However, fast transcoding with parallel bit streams requires only 3.5 cycles per byte for our German text, 6.2 cycles per byte for the Japanese text and a mere 0.9 cycles per byte for ASCII-based data files [4]. Thus, adjusting for the overhead of transcoding does not substantially alter the performance ratios of Parabix vs. Xerces.

It is also interesting to compare these results with the reported results for XML Screamer XML Screamer is a fast XML parser that demonstrates the performance that may be achieved with schema-dependent compilation together with tight integration across levels in the parser architecture. XML Screamer is reported to achieve validating parsing and object creation (according to XML Schema, but not DTDs) from 23 to 46 megabytes/second per processor GigaHertz. This corresponds to 22 to 43 cycles per byte which compares quite favorably with the Expat performance reported here, given that XML Screamer is carrying out the additional work of validation and object creation. An important conclusion from the XML Screamer work is that validation need not impose a performance penalty; it can instead provide useful information for parser optimization. Employing the techniques of XML Screamer as validation is integrated into Parabix should be similarly beneficial.

### 3.6 Data Cache Behaviour

An important determinant of performance in modern processors is how memory is managed to ensure that necessary data is available in the processor caches when needed. A small level one (L1) cache can typically be accessed within a few CPU cycles, while a larger level two cache (L2) may have an additional latency on the order of 10 cycles. However, when L2 cache misses require that memory be accessed to pull in the necessary data, a latency in tens or even hundreds of cycles is typical, depending on memory characteristics.

Figures 4 and 5 show the L1 and L2 data cache behaviour, respectively, in terms of cache misses per byte of XML input. These figures show that Expat has the best L1 cache performance, while Parabix has dramatically better L2 cache performance than either Expat

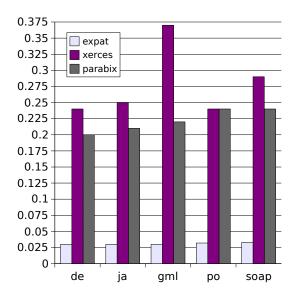


Figure 4: Level 1 Data Cache Misses Per Byte

or Xerces. The buffering strategy of Parabix may explain both the poorer L1 cache miss rate and the superior L2 cache miss rate. Parabix initially loads XML byte data into an internal buffer 100K at a time, and then performs transposition to parallel bit streams and bit stream computations over this buffer. The 100K buffer is too large to fit into L1 cache, but is sized so that it and data derived from it easily fits into the L2 cache. Furthermore, the initial load and transpose phase has a purely sequential memory access pattern that should support automatic prefetching by the processor's built-in hardware prefetch unit. In contrast, automatic prefetching is made difficult with the interleaved memory access patterns typical of byte-at-a-time processing: loading a few bytes from the input buffer, validating character input with memory tables in another area, looking up symbols in yet another and so on. With an L2 cache latency of 200 cycles per byte, for example, Expat and Xerces may each be paying a 6 cycle per byte penalty based on their L2 cache miss rates, while Parabix has an insignificant cost. Nevertheless there may be room for improvement in the L1 cache access of Parabix, which may cost about 2 cycles per byte based on a 10 cycles per byte penalty for

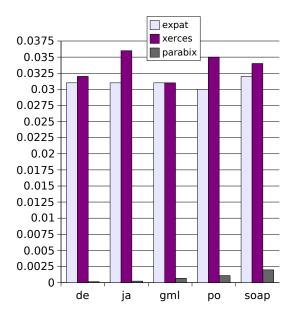


Figure 5: Level 2 Data Cache Misses Per Byte

L1 cache misses.

# 3.7 Branching Behaviour

In contemporary processors, the success rate of branch prediction is a further key factor in determining overall application performance. Branch prediction is essential to obtain maximum benefit from speculative execution and pipelining. In text-based applications, branch prediction can be very difficult due to the variable-length nature of syntactic elements and the multiway branching that is often needed when several syntactic alternatives are legal at particular locations. Thus branch mispredictions can be expected to be quite frequent. A cost of up to 20 CPU cycles is quite common for each branch misprediction.

Figure 6 show the total number of conditional branch instructions executed per byte for each of the parsers while Figure 7 shows the number of branch mispredictions per byte. Parabix has a dramatically lower branch misprediction rate, based on a substantially lower use of conditional branching overall. Each of the parallel bit stream phases within parabix are essentially branch-free with only a few branches per 128 byte block. The net effect is

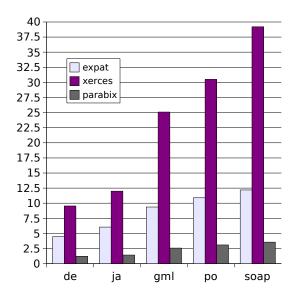


Figure 6: Conditional Branches Per Byte

that cost of branch misprediction is very low in Parabix, whereas the overall branch misprediction in each of Expat and Xerces is expected to be several cycles per byte.

# 3.8 Cycles Per Instruction

Superscalar processors have the capability to issue and discharge multiple instructions per cycle on modern processors. A measure of how well a particular application makes use of processor resources is therefore the average number of CPU cycles required per instruction. Figure 8 shows the performance of each of the parsers on this metric. Parabix performs substantially better than either Expat or Xerces, making substantially better use of the superscalar capabilities. This is likely due to the long sections of straight-line code that are employed within the parallel bit stream calculations, enabling the processor to take better advantage of its instruction scheduling and out-of-order execution capabilities.

As processor technology advances, applications that have better performance with respect to cache behaviour, branch prediction and superscalar execution should be in a better position to take advantage of the increased

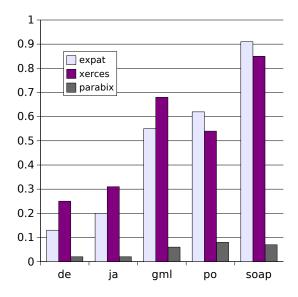


Figure 7: Branch Mispredictions Per Byte

processor capabilities. Thus, it is quite reasonable to expect that the relative performance advantage of parallel bit stream technology over byte-at-a-time code to increase with continued processor evolution.

# 4 Analyzing Parabix

This section addresses the current and prospective performance characteristics of Parabix in more detail. The first subsection addresses current performance broken down by functional component. The second subsection then goes on to briefly discuss prospective performance characteristics emphasis the adaptation of Parabix for multicore processors.

# 4.1 Component Performance

To gain a finer-grained understanding of how the overall performance of parabix is affected by the use of parallel bit stream technology and the performance of particular functional components, the PAPI performance counters were further employed in a series of experiments to assess the performance of individual components. The execution time figures are presented

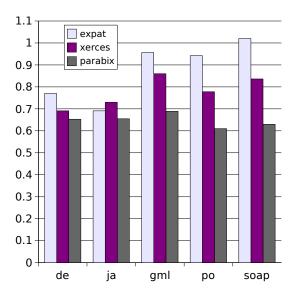


Figure 8: Cycles Per Instruction

in Table 4 in terms of CPU cycles per input byte.

The first four rows of Table 4 show functionality primarily related to parallel bit stream computation. Each of the functions is handled by a separate parsing method that completes the analysis for a full buffer of data (approximately 100K in the current set up) before control is passed to the next phase. This permits direct instrumentation using the PAPI counters to measure performance on a buffer by buffer basis.

The first row of Table 4, labeled S2P, is the execution time required for the serial-toparallel transposition step, that is, the conversion of byte stream data to the parallel bit stream representation. Regardless of input file characteristic, this step requires 1.1 cycles per input byte on 64-bit architecture. This is one area in which there was a clear and repeatable difference between performance in 32-bit mode versus 64-bit mode, with S2P performance consistently measuring at 1.2 cycles per input mode in the latter case. The improvement in 64-bit mode is presumably because of the increase in available SIMD registers from 8 to 16, thereby allowing the compiler to eliminate the stacking of some temporaries used in

Row	Component	de	ja	gml	ро	soap
1	S2P	1.08	1.09	1.09	1.08	1.08
2	UTF-8 Validation	0.33	0.52	0.11	0.13	0.12
3	WS/Control	0.26	0.26	0.26	0.26	0.26
4	Lexical Streams	1.00	1.00	1.01	1.00	1.01
5	Parsing	2.86	3.25	3.02	4.71	4.92
6	Semantics	0.57	0.88	4.82	6.22	7.41
7	Total	6.1	7.0	10.3	13.4	14.8

Table 4: Parabix Component Performance (Cycles Per Byte)

the calculations.

The second row of the table shows the time required for validating the correctness of UTF-8 characters (as well as proving the absence of 0xFFFE and 0xFFFF, illegal in XML). When the input data contains a substantial proportion of 3-byte UTF-8 sequences, the validation cost is about 0.5 cycles per byte, as shown in the case of the Japanese language wiki document. On the other hand, for data-oriented documents primarily confined to the ASCII repertoire, only about 0.1 cycles per byte is used. The GML documents fall into this category.

The third row of Table 4 shows the time spent calculating the whitespace lexical item stream together with validation that restricted control characters are not permitted. These functions require different bit stream processing depending on XML version (1.0 or 1.1), and hence have been separated out from the other validation and lexical item processing calculations, which are version independent. These functions consistently require about one quarter of a cycle per byte as presently implemented.

The fourth row of the table shows the time for calculating all the core lexical item streams needed by the parsing engine. These are the stream identifying the positions of markup when scanning text, the positions of name follow characters when scanning names, the positions of quoted value terminators when scanning attribute values, the positions of hyphens and question marks for scanning comments and processing instructions, the positions of possible CDATA end markers for scanning CDATA sections and text, as well as streams for dec-

imal and hexadecimal numerals used in scanning character references. The total time required for all of these stream calculations together comes to one cycle per byte, independent of input file characteristics.

As shown, parallel bit stream calculations overall account for well less than 3 cycles per byte. In fact, there are prospects for improving the performance of these functions by more careful coding. More importantly, however, these functions all comprise primarily straightline SIMD code in predictable iteration patterns. As processor SIMD capabilities improve, the performance of these functions will naturally improve. This has already been observed in the move from Pentium 4 technology to Core 2.

The fifth row of Table 4, labeled Parsing, shows the time attributed to making use of the lexical item stream data to parse the contextfree syntactic structure of the XML input document. These figures were calculated by disabling semantic actions including all symbol table lookup and insertion and associated XML rules. The parsing time was then determined by subtracting the time for parallel bit stream formation (sum of rows 1 to 4) from the total time for this simplified parser. Parsing time is smaller for the document-oriented XML files primarily because there are longer runs of text content. In either case, however, the parsing time is a small fraction of what would be expected in a byte-at-a-time processor, clearly demonstrating the value of replacing byte-ata-time loops with bit scans.

The sixth row of the table then shows the time attributed to semantic processing, including symbol table lookup actions for every instance of an element, attribute, entity or notation name encountered in the XML data. Technically, the need for symbol lookup is limited in a nonvalidating processor, particularly if there is no DTD. However, symbol lookup is a convenient way to validate the syntax of XML names [8] and hence avoid the cost of character-by-character lookup for each instance other than the first. Furthermore, the symbol table system is an essential component of a full XML processor that supports both validation and semantics-based applications.

Presently, semantic processing accounts for half or more of the total processing times shown in row 7. However, there may be considerable room for improvement here. The current version of parabix does not employ any parallel bit stream or other SIMD techniques in symbol table processing, relying instead on the built-in hash table functionality of the C++ standard template library. Work in progress includes parallel hash function computation, length-sorted multipass hashing to avoid branch penalties as as well as assessment of SIMD cuckoo hashing [14] as potential techniques to further improve performance.

# 4.2 Parabix on Multicore

An important goal of ongoing work is to leverage the inherent parallelism of parallel bit stream technology to take advantage of parallel processing on multicore processors.

The core bitstream algorithms are expected to be highly parallelizable. For example, a large file may be partitioned into 128K chunks, each chunk being assigned to a separate core. Transposition to parallel bit stream form may be carried out completely independently on each chunk. Character validation, transcoding and lexical item stream computation can also be carried out with a small overlap at the partition boundaries. The overlap is needed to deal with the possibility that multibyte UTF-8 characters and XML delimiters may cross partition boundaries. However, the extent of influence of computations from one partition to the next is strictly limited. This allows the computations in each partition to proceed in parallel with either a small preprocessing or postprocessing phase to deal with the partition boundary.

Parsing, however, is an inherently sequential activity. To partition appropriately at a boundary would require knowing the parsing state at the boundary; but the influence on this state may reach back to the beginning of the file. Nevertheless, there are some reasonable alternatives to consider distribution of parsing activities. One is to use a lightweight preparse of the entire file identifying the extent of all comments, processing instructions and CDATA sections. Typically, there are very few of these, allowing scanning to be completed very quickly. An "excluded content" bitstream can then be formed as the union of the extent of all of these markups. The properties of XML content (after any prolog) mean that all remaining opening angle brackets can only legally represent the beginning of a tag, while all remaining ampersands can only legally represent the beginning of a reference. This would allow the process of identifying the positions of these items to be processed independently and in parallel in each of the partitions. Subsequently, the identification and hashing of element and entity names could also be handled independently and parallel. Assessment of this kind of approach is an area of ongoing research.

### 5 Conclusions

Parallel bit stream technology offers the prospect of dramatic performance improvement in text processing applications as illustrated by the example of the parabix XML parser considered in this paper. On today's commodity processors, the performance benefits are substantial. On future processors, the advantage over byte-at-a-time processing may be expected to increase with improved SIMD support as well as with data parallel parsing distributed over multiple cores.

The performance improvement arises from the idea of simultaneously processing characters 128 at a time with 128-bit registers. This parallelized approach works well with modern pipelined processors and allows text processing applications to proceed with a minimum of mispredicted branch instructions and L2 cache misses.

Parabix itself is a continuing focus of re-

search and development as an open source project. From a research perspective, our goal is to develop and assess software engineering techniques that can deliver high-performance, portability and demonstrable quality in production systems based on parallel bit stream technology. From a development perspective, our goal is to enable the widespread deployment of parabix throughout computing infrastructure. Patents have been filed to allow this work to be carried out through an open source framework using a patentleft commercialization strategy.

# Acknowledgements

This work was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

# About the Authors

Robert D. Cameron earned his Ph.D from the University of British Columbia in 1983. He is presently a Professor of Computing Science at Simon Fraser University with research interests in programming languages, software engineering, data compression, digital libraries and sociotechnical design of public computing infrastructure. Since completing a six-year term as Associate Dean of Applied Sciences in 2004, his research focus has been on high-performance XML processing using the SIMD capabilities of commodity processors.

Kenneth S. Herdy completed an Advanced Diploma of Technology in Geographical Information Systems through the British Columbia Institute of Technology in 2003 and earned a Bachelor of Science in Computing Science with a Certificate in Spatial Information Systems at Simon Fraser University in 2005. He is currently pursuing graduate studies in Computing Science at Simon Fraser University. His research involves an analysis of the principal techniques that may be used to improve XML processing performance in the context of the Geography Markup Language (GML).

Dan Lin earned a B.Sc. degree in Computer Science from Beihang University in 2006. She

is presently working towards an M.Sc. in Computing Science at Simon Fraser University. Her present research deals with high-performance text processing using SIMD techniques in network applications such as intrusion detection, virus scanning and XML processing.

### References

- [1] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.
- [2] Xerces C++ Parser. http://xerces.apache.org/xerces-c/.
- [3] Cameron, Robert D. u8u16 A High-Speed UTF-8 to UTF-16 Transcoder Using Parallel Bit Streams. Technical Report TR 2007-18, Simon Fraser University, Burnaby, BC, Canada, 2007.
- [4] Cameron, Robert D. A Case Study in SIMD Text Processing with Parallel Bit Streams. In ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), Salt Lake City, Utah, 2008.
- [5] Clark, James. The Expat XML Parser. http://expat.sourceforge.net/.
- [6] DuCharme, Bob. Documents vs. data, schemas vs. schemas. In XML 2004, Washington D.C., 2004.
- [7] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual, 2005.
- [8] Kostoulas, M. G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., and Mercaldi, M. XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization. In Proceedings of the 15th International Conference on World Wide Web (WWW '06), pages 93-102, 2006.
- [9] Nicola, Matthias, and John, Jasmi. XML Parsing: A Threat to Database Performance. In Proceedings of the Twelfth International Conference on Information and Knowledge Management, New Orleans, Louisiana, 2003.

- [10] Perkins, E., Kostoulas, M., Heifets, A., Matsa, M., and Mendelsohn, N. Performance Analysis of XML APIs. In XML 2005, Atlanta, Georgia, November 2005.
- [11] Pettersson, Michael. Linux x86 Performance-Monitoring Counters Driver. http://user.it.uu.se/mikpe/linux/perfctr.
- [12] Psaila, Giuseppe. On the Problem of Coupling Java Algorithms and XML Parsers. In 17th International Conference on Database and Expert Systems Applications (DEXA'06), pages 487–491, 2006.
- [13] Ron Lake, David Burggraf, Milan Trninic, and Laurie Rae. Geography Mark-Up Language: Foundation for the Geo-Web, pages 3–4. John Wiley & Sons, Inc., 2004.
- [14] Ross, Kenneth A. Efficient Hash Probes on Modern Processors. In *Proceedings* of the 23rd International Conference on Data Engineering (ICDE 2007), Istanbul, Turkey, 2007.
- [15] Zhao, Li Laxmi Bhuyan. Performance Evaluation and Acceleration for XML Data Parsing. In 9th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), Austin, Texas, 2006.