Parallel Scanning with Bitstream Addition: An XML Case Study

Robert D. Cameron, Ehsan Amiri, Kenneth S. Herdy, Dan Lin,
Thomas C. Shermer, and Fred P. Popowich

Simon Fraser University, Surrey, BC, Canada
{cameron,eamiri,ksherdy,lindanl,shermer,popowich}@cs.sfu.ca

Abstract. A parallel scanning method using the concept of bitstream addition is introduced and studied in application to the problem of XML parsing and well-formedness checking. On processors supporting W-bit addition operations, the method can perform up to W finite state transitions per instruction. The method is based on the concept of parallel bitstream technology, in which parallel streams of bits are formed such that each stream comprises bits in one-to-one correspondence with the character code units of a source data stream. Parsing routines are initially prototyped in Python using its native support for unbounded integers to represent arbitrary-length bitstreams. A compiler then translates the Python code into low-level C-based implementations. These low-level implementations take advantage of the SIMD (single-instruction multiple-data) capabilities of commodity processors to yield a dramatic speed-up over traditional alternatives employing byte-at-a-time parsing.

Keywords: SIMD text processing, parallel bitstreams, XML, parsing.

1 Introduction

Although the finite state machine methods used in the scanning and parsing of text streams is considered to be the hardest of the "13 dwarves" to parallelize [1], parallel bitstream technology shows considerable promise for these types of applications [3,4]. In this approach, character streams are processed N positions at a time using the N-bit SIMD registers commonly found on commodity processors (e.g., 128-bit XMM registers on Intel/AMD chips). This is achieved by first slicing the byte streams into eight separate basis bitstreams, one for each bit position within the byte. These basis bitstreams are then combined with bitwise logic and shifting operations to compute further parallel bit streams of interest, such as the `[<]` bit stream marking the position of all opening angle brackets in an XML document.

Using these techniques as well as the *bit scan* instructions also available on commodity processors, the Parabix 1 XML parser was shown to considerably accelerate XML parsing in comparison with conventional byte-at-a-time parsers in applications such as statistics gathering [4] and as GML to SVG conversion [6]. Other efforts to accelerate XML parsing include the use of custom XML

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 2–13, 2011.

chips [8], FPGAs [5], careful coding and schema-based processing[7] and multi-thread/multicore speedups based on data parallelism[9,10].

In this paper, we further increase the parallelism in our methods by introducing a new parallel scanning primitive using bitstream addition. In essence, this primitive replaces the sequential bit scan operations underlying Parabix 1 with a new approach that independently advances multiple marker bits in parallel using simple addition and logic operations. This paper documents the technique and evaluates it in application to the problem of XML parsing and well-formedness checking.

Section 2 reviews the basics of parallel bitstream technology and introduces our new parallel scanning primitive. Section 3 goes on to show how this primitive may be used in XML scanning and parsing, while Section 4 discusses the construction of a complete XML well-formedness checker based on these techniques. Section 5 then briefly describes the compiler technology used to generate the low level code for our approach. A performance study in Section 6 shows that the new Parabix 2 parser is dramatically faster than traditional byte-at-a-time parsers as well as the original Parabix 1 parser, particularly for dense XML markup. Section 7 concludes the paper.

2 The Parallel Bitstream Method

2.1 Fundamentals

A bitstream is simply a sequence of 0s and 1s, where there is one such bit in the bitstream for each character in a source data stream. For parsing, and other text processing tasks, we need to consider multiple properties of characters at different stages during the parsing process. A bitstream can be associated with each of these properties, and hence there will be multiple (parallel) bitstreams associated with a source data stream of characters.

The starting point for bitstream methods are *basis* bitstreams and their use in determining *character-class* bitstreams. The kth basis bitstream B_k consists of the kth bit (0-based, starting at the the least significant bit) of each character

```
source data ◁ ----173942---654----1----49731----321--
        B7    ..................................
        B6    ..................................
        B5    11111111111111111111111111111111111
        B4    ....111111...111....1....11111....111..
        B3    1111...1..111...1111.1111.1...1111...11
        B2    1111.1..1.1111111111.11111.1..1111...11
        B1    .....11..1...1.............11.....11...
        B0    11111111..111.1.111111111.111111111.111
     [0-9]    ....111111...111....1....11111....111..
```

Fig. 1. Basis and Character-Class Bitstreams

in the source data stream; thus each B_k is dependent on the encoding of the source characters (ASCII, UTF-8, UTF-16, etc.). Given these basis bitstreams, it is then possible to combine them using bitwise logic in order to compute character-class bitstreams, that is, streams that identify the positions at which characters belonging to a particular class occur. For example, the character class bitstream $D =$ `[0-9]` marks with 1s the positions at which decimal digits occur. These bitstreams are illustrated in Figure 1, for an example source data stream consisting of digits and hyphens. This figure also illustrates some of our conventions for figures: the left triangle $\lhd$ after "source data" indicates that all streams are read from right to left (i.e., they are in little-endian notation). We also use hyphens in the input stream represent any character that is not relevant to a character class under consideration, so that relevant characters stand out. Furthermore, the 0 bits in the bitstreams are represented by periods, so that the 1 bits stand out.

Transposition of source data to basis bitstreams and calculation of character-class streams in this way is an overhead on parallel bit stream applications, in general. However, using the SIMD capabilities of current commodity processors, these operations are fast, with an amortized overhead of about 1 CPU cycle per byte for transposition and less than 1 CPU cycle per byte for all the character classes needed for XML parsing [4].

Beyond the bitwise logic needed for character class determination, we also need *upshifting* to deal with sequential combination. The upshift $n(S)$ of a bitstream S is obtained by shifting the bits in S one position forward, then placing a 0 bit in the starting position of the bitstream; n is meant to be mnemonic of "next". In $n(S)$, the last bit of S may be eliminated or retained for error-testing purposes.

2.2 A Parallel Scanning Primitive

In this section, we introduce the principal new feature of the paper, a parallel scanning method based on bitstream addition. Key to this method is the concept of *marker* bitstreams. Marker bitstreams are used to represent positions of interest in the scanning or parsing of a source data stream. The appearance of a 1 at a position in a marker bitstream could, for example, denote the starting position of an XML tag in the data stream. In general, the set of bit positions in a marker bitstream may be considered to be the current parsing positions of multiple parses taking place in parallel throughout the source data stream.

Figure 2 illustrates the basic concept underlying parallel parsing with bitstream addition. All streams are shown in little-endian representation, with streams reading from right-to-left. The first row shows a source data stream that includes several spans of digits, together with other nondigit characters shown as hyphens. The second row specifies the parsing problem using a marker bitstream M_0 to mark four initial marker positions. In three instances, these markers are at the beginning (i.e., little end) of a span, while one is in the middle of a span. The parallel parsing task is to move each of the four markers forward (to the left) through the corresponding spans of digits to the immediately following positions.

source data $\lhd$	`----173942---654----1----49731----321--`
M_0	`.........1.....1....1......1...........`
$D =$ `[0-9]`	`....111111...111....1....11111....111..`
$M_0 + D$	`...1........1......1....1...11....111..`
$M_1 = (M_0 + D) \wedge \neg D$	`...1........1......1....1.............`

Fig. 2. Parallel Scan Using Bitstream Addition and Mask

The third row of Figure 2 shows the derived character-class bitstream D identifying positions of all digits in the source stream. The fourth row then illustrates the key concept: marker movement is achieved by binary addition of the marker and character class bitstreams. As a marker 1 bit is combined using binary addition to a span of 1s, each 1 in the span becomes 0, generating a carry to add to the next position to the left. For each such span, the process terminates at the left end of the span, generating a 1 bit in the immediately following position. These generated 1 bits represent the moved marker bits. However, the result of the addition also produces some additional bits that are not involved in the scan operation. These are easily removed as shown in the fifth row, by applying bitwise logic to mask off any bits from the digit bitstream; these can never be marker positions resulting from a scan. The addition and masking technique allows matching of the regular expression `[0-9]*` for any reasonable (conflict-free) set of initial markers specified in M_0.

In the remainder of this paper, the notation $s(M, C)$ denotes the operation to scan from an initial set of marker positions M through the spans of characters belonging to a character class C found at each position.

$$s(M, C) = (M + C) \wedge \neg C$$

3 XML Scanning and Parsing

We now consider how the parallel scanning primitive can be applied to the following problems in scanning and parsing of XML structures: (1) parallel scanning of XML decimal character references, and (2) parallel parsing of XML start tags. The grammar of these structures is shown in Figure 3.

DecRef ::=	'`&#`' Digit$^+$ '`;`'
Digit ::=	`[0-9]`
STag ::=	'`<`' Name (W Attribute)* W$^?$ '`>`'
Attribute ::=	Name W$^?$ '=' W$^?$ AttValue
AttValue ::=	('`"`' `[^<"]*` '`"`') | ("'" `[^<']*` "'")
W ::=	(`\x20` | `\x9` | `\xD` | `\xA`)$^+$

Fig. 3. XML Grammar: Decimal Character References and Start Tags

Figure 4 shows the parallel parsing of decimal references together with error checking. For clarity, the streams are now shown in left-to-right order as indicated by the ▷ symbol. The source data includes four instances of potential decimal references beginning with the `&` character. Of these, only the first one is legal according to the decimal reference syntax, the other three instances are in error. These references may be parsed in parallel as follows. The starting marker bitstream M_0 is formed from the `[&]` character-class bitstream as shown in the second row. The next row shows the result of the marker advance operation $n(M_0)$ to produce the new marker bitstream M_1. At this point, the grammar requires a hash mark, so the first error bitstream E_0 is formed using a bitwise "and" operation combined with negation, to indicate violations of this condition. Marker bitstream M_2 is then defined as those positions immediately following any M_1 positions not in error. In the following row, the condition that at least one digit is required is checked to produce error bitstream E_1. A parallel scan operation is then applied through the digit sequences as shown in the next row to produce marker bitstream M_3. The final error bitstream E_2 is produced to identify any references without a closing semicolon. In the penultimate row, the final marker bitstream M_4 marks the positions of all fully-checked decimal references, while the last row defines a unified error bitstream E indicating the positions of all detected errors.

Initialization of marker streams may be achieved in various ways, dependent on the task at hand. In the XML parsing context, we rely on an important property of well-formed XML: after an initial filtering pass to identify XML comments, processing instructions and CDATA sections, every remaining `<` in the file must be the initial character of a start, end or empty element tag, and every remaining `&` must be the initial character of a general entity or character reference. These assumptions permit easy creation of marker bitstreams for XML tags and XML references.

The parsing of XML start tags is a richer problem, involving sequential structure of attribute-value pairs as shown in Figure 3. Using the bitstream addition technique, our method is to start with the opening angle bracket of all tags as the initial marker bitstream for parsing the tags in parallel, advance through the element name and then use an iterative process to move through attribute-value pairs.

source data ▷	`-&#978;-&9;--&#;--&#13!-`
M_0	`.1.....1....1....1.....`
$M_1 = n(M_0)$	`..1.....1....1....1....`
$E_0 = M_1 \wedge \neg$`[#]`	`.........1.............`
$M_2 = n(M_1 \wedge \neg E_0)$	`...1..........1....1...`
$E_1 = M_2 \wedge \neg D$	`..............1........`
$M_3 = s(M_2 \wedge \neg E_1, D)$	`......1..............1.`
$E_2 = M_3 \wedge \neg$`[;]`	`......................1.`
$M_4 = M_3 \wedge \neg E_2$	`......1.................`
$E = E_0 \mid E_1 \mid E_2$	`.........1.....1......1.`

Fig. 4. Parsing Decimal References

Figure 5 illustrates the parallel parsing of three XML start tags. The figure omits determination of error bitstreams, processing of single-quoted attribute values and handling of empty element tags, for simplicity. In this figure, the first four rows show the source data and three character class bitstreams: N for characters permitted in XML names, W for whitespace characters, and Q for characters permitted within a double-quoted attribute value string.

```
source data ▷              --<e a= "137">---<el2 a="17" a2="3379">---<x>--
N = name chars             11.1.1...111..111.111.1..11..11..1111..111.1.11
W = white space            ....1..1.............1......1..................
Q = ¬["<]                  11.11111.111.1111.111111.11.1111.1111.1111.1111

M0                         ..1.............1.........................1....
M1 = n(M0)                 ...1.............1.........................1...
M0,7 = s(M1, N)            ....1.............1.........................1..
M0,8 = s(M0,7, W) ∧ ¬[>]   .....1.............1.........................

M1,1 = s(M0,8, N)          ......1.............1.........................
M1,2 = s(M1,1, W)∧[=]      ......1.............1.........................
M1,3 = n(M1,2)             .......1.............1........................
M1,4 = s(M1,3, W)∧["]      ........1.............1.......................
M1,5 = n(M1,4)             .........1.............1......................
M1,6 = s(M1,5, Q)∧["]      ...........1.............1....................
M1,7 = n(M1,6)             ............1.............1...................
M1,8 = s(M1,7, W) ∧ ¬[>]   ............................1..................

M2,1 = s(M1,8, N)          ................................1..............
M2,2 = s(M2,1, W)∧[=]      ................................1..............
M2,3 = n(M2,2)             .................................1.............
M2,4 = s(M2,3, W)∧["]      .................................1.............
M2,5 = n(M2,4)             ..................................1............
M2,6 = s(M2,5, Q)∧["]      ....................................1........
M2,7 = n(M2,6)             .....................................1........
M2,8 = s(M2,7, W) ∧ ¬[>]   ..........................................
```

Fig. 5. Start Tag Parsing

The parsing process is illustrated in the remaining rows of the figure. Each successive row shows the set of parsing markers as they advance in parallel using bitwise logic and addition. Overall, the sets of marker transitions can be divided into three groups.

The first group M_0 through $M_{0,8}$ shows the initiation of parsing for each of the tags through the opening angle brackets and the element names, up to the first attribute name, if present. Note that there are no attribute names in the final tag shown, so the corresponding marker becomes zeroed out at the closing angle bracket. Since $M_{0,8}$ is not all 0s, the parsing continues.

The second group of marker transitions $M_{1,1}$ through $M_{1,8}$ deal with the parallel parsing of the first attribute-value pair of the remaining tags. After these operations, there are no more attributes in the first tag, so its corresponding marker becomes zeroed out. However, $M_{1,8}$ is not all 0s, as the second tag still has an unparsed attribute-value pair. Thus, the parsing continues.

The third group of marker transitions $M_{2,1}$ through $M_{2,8}$ deal with the parsing of the second attribute-value pair of this tag. The final transition to $M_{2,8}$ shows the zeroing out of all remaining markers once two iterations of attribute-value processing have taken place. Since $M_{2,8}$ is all 0s, start tag parsing stops.

The implementation of start tag processing uses a while loop that terminates when the set of active markers becomes zero, i.e. when some $M_{k,8} = 0$. Considered as an iteration over unbounded bitstreams, all start tags in the document are processed in parallel, using a number of iterations equal to the maximum number of attribute-value pairs in any one tag in the document. However, in block-by-block processing, the cost of iteration is considerably reduced; the iteration for each block only requires as many steps as there are attribute-value pairs overlapping the block.

Following the pattern shown here, the remaining syntactic features of XML markup can similarly be parsed with bitstream based methods. One complication is that the parsing of comments, CDATA sections and processing instructions must be performed first to determine those regions of text within which ordinary XML markups are not parsed (i.e., within each of these types of construct. This is handled by first parsing these structures and then forming a *mask bitstream*, that is, a stream that identifies spans of text to be excluded from parsing (comment and CDATA interiors, parameter text to processing instructions).

4 XML Well-Formedness

In this section, we consider the full application of the parsing techniques of the previous section to the problem of XML well-formedness checking [2]. We look not only at the question of well-formedness, but also at the identification of error positions in documents that are not well-formed.

Most of the requirements of XML well-formedness checking can be implemented using two particular types of computed bitstream: *error bitstreams*, introduced in the previous section, and *error-check bitstreams*. Recall that an error bitstream stream is a stream marking the location of definite errors in accordance with a particular requirement. For example, the E_0, E_1, and E_2 bitstreams as computed during parsing of decimal character references in Figure 4 are error bitstreams. One bits mark definite errors and zero bits mark the absence of an error. Thus the complete absence of errors according to the requirements listed may be determined by forming the bitwise logical "or" of these bitstreams and confirming that the resulting value is zero. An error check bitstream is one that marks potential errors to be further checked in some fashion during post-bitstream processing. An example is the bitstream marking the start positions

of CDATA sections. This is a useful information stream computed during bitstream processing to identify opening `<![` sequences, but also marks positions to subsequently check for the complete opening delimiter `<![CDATA[` at each position.

In typical documents, most of these error-check streams will be quite sparse or even zero. Many error conditions could actually be fully implemented using bitstream techniques, but at the cost of a number of additional logical and shift operations. In general, the conditions are easier and more efficient to check one-at-a-time using multibyte comparisons on the original source data stream. With very sparse streams, it is very unlikely that multiple instances occur within any given block, thus eliminating the benefit of parallel evaluation of the logic.

The requirement for name checking merits comment. XML names may use a wide range of Unicode character values. It is too expensive to check every instance of an XML name against the full range of possible values. However, it is possible and inexpensive to use parallel bitstream techniques to verify that any ASCII characters within a name are indeed legal name start characters or name characters. Furthermore, the characters that may legally follow a name in XML are confined to the ASCII range. This makes it useful to define a name scan character class to include all the legal ASCII characters for names as well as all non-ASCII characters. A namecheck character class bitstream will then be defined to identify non-ASCII characters found within namescans. In most documents this bitstream will be all 0s; even in documents with substantial internationalized content, the tag and attribute names used to define the document schema tend to be confined to the ASCII repertoire. In the case that this bitstream is nonempty, the positions of all 1 bits in this bitstream denote characters that need to be individually validated.

Attribute names within a single XML start tag or empty element tag must be unique. This requirement could be implemented using one of several different approaches. Standard approaches include: sequential search, symbol lookup, and Bloom filters [5].

Except for empty element tags, XML tags come in pairs with names that must be matched. To discharge this requirement, we form a bitstream consisting of the disjunction of three bitstreams formed during parsing: the bitstream marking the positions of start or empty tags (which have a common initial structure), the bitstream marking tags that end using the empty tag syntax (“`/>`”), and the bitstream marking the occurrences of end tags. In post-bitstream processing, we iterate through this computed bitstream and match tags using an iterative stack-based approach.

An XML document consists of a single root element within which all others contained; this constraint is also checked during post-bitstream processing. In addition, we define the necessary ”miscellaneous” bitstreams for checking the prolog and epilog material before and after the root element.

Overall, parallel bitstream techniques are well-suited to verification problems such as XML well-formedness checking. Many of the character validation and syntax checking requirements can be conveniently and efficiently implemented

using error streams. Other requirements are also supported by the computation of error-check streams for simple post-bitstream processing or composite stream over which iterative stack-based procedures can be defined for checking recursive syntax. To assess the completness of our analysis, we have confirmed that our implementations correctly handle all the well-formedness checks of the W3C XML Conformance Test Suite.

5 Compilation to Block-Based Processing

While our Python implementation of the techniques described in the previous section works on unbounded bitstreams, a corresponding C implementation needs to process an input stream in blocks of size equal to the SIMD register width of the processor it runs on. So, to convert Python code into C, the key question becomes how to transfer information from one block to the next.

The answer lies in the use of *carry bits*. The parallel scanning primitive uses only addition and bitwise logic. The logic operations do not require information flow accross block boundaries, so the information flow is entirely accounted by the carry bits for addition. Carry bits also capture the information flow associated with upshift operations, which move information forward one position in the file. In essence, an upshift by one position for a bitstream is equivalent to the addition of the stream to itself; the bit shifted out in an upshift is in this case equivalent to the carry generated by the additon.

Properly determining, initializing and inserting carry bits into a block-by-block implementation of parallel bitstream code is a task too tedious for manual implementation. We have thus developed compiler technology to automatically insert declarations, initializations and carry save/restore operations into appropriate locations when translating Python operations on unbounded bitstreams into the equivalent low-level C code implemented on a block-by-block bases. Our current compiler toolkit is capable of inserting carry logic using a variety of strategies, including both simulated carry bit processing with SIMD registers, as well as carry-flag processing using the processor general purpose registers and ALU. Details are beyond the scope of this paper, but are described in the on-line source code repository at parabix.costar.sfu.ca.

6 Performance Results

In this section, we compare the performance of our `xmlwf` implementation using the Parabix 2 technology described above with several other implementations. These include the original `xmlwf` distributed as an example application of the `expat` XML parser, implementations based on the widely used Xerces open source parser using both SAX and DOM interfaces, and an implementation using our prior Parabix 1 technology with bit scan operations.

Table 1 shows the document characteristics of the XML instances selected for this performance study, including both document-oriented and data-oriented XML files. The jawiki.xml and dewiki.xml XML files are document-oriented XML instances of Wikimedia books, written in Japanese and German, respectively. The remaining files are data-oriented. The roads.gml file is an instance of Geography Markup Language (GML), a modeling language for geographic information systems as well as an open interchange format for geographic transactions on the Internet. 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.

Table 1. XML Document Characteristics

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 2 shows performance measurements for the various `xmlwf` implementations applied to the test suite. Measurements are made on a single core of an Intel Core 2 system running a stock 64-bit Ubuntu 10.10 operating system, with all applications compiled with llvm-gcc 4.4.5 optimization level 3. Measurements are reported in CPU cycles per input byte of the XML data files in each case. The first row shows the performance of the Xerces C parser using the tree-building DOM interface. Note that the performance varies considerably depending on markup density. Note also that the DOM tree construction overhead is substantial and unnecessary for XML well-formedness checking. Using the event-based SAX interface to Xerces gives much better results as shown in the second row. The third row shows the best performance of our byte-at-a-time parsers, using the original `xmlwf` based on expat.

The remaining rows of Table 2 show performance of parallel bitstream implementations, including post-bitstream processing. The first row shows the performance of our Parabix 1 implementation using bit scan instructions. While showing a substantial speed-up over the byte-at-a-time parsers in every case, note also that the performance advantage increases with increasing markup density, as expected. The last two rows show Parabix 2 implementations using different carry-handling strategies, with the "simd" row referring to carry computations performed with simulated calculation of propagated and generated carries using SIMD operations, while the "adc64" row referring to an implementation directly employing the processor carry flags and add-with-carry instructions on 64-bit

Table 2. Parser Performance (Cycles Per Byte)

Parser Class	Parser	dewiki.xml	jawiki.xml	roads.gml	po.xml	soap.xml
Byte at-a Time	Xerces (DOM)	37.921	40.559	72.78	105.497	125.929
	Xerces (SAX)	19.829	24.883	33.435	46.891	57.119
	expat	12.639	16.535	32.717	42.982	51.468
Parallel Bit Stream	Parabix1	8.313	9.335	13.345	16.136	19.047
	Parabix2 (simd)	6.103	6.445	8.034	8.685	9.53
	Parabix2 (adc64)	5.123	5.996	6.852	7.648	8.275

general registers. In both cases, the overall performance is impressive, with the increased parallelism of parallel bit scans clearly paying off in improved performance for dense markup.

7 Conclusion

In application to the problem of XML parsing and well-formedness checking, the method of parallel parsing with bitstream addition is effective and efficient. Using only bitstream addition and bitwise logic, it is possible to handle all of the character validation, lexical recognition and parsing problems except for the recursive aspects of start and end tag matching. Error checking is elegantly supported through the use of error streams that eliminate separate if-statements to check for errors with each byte. The techniques are generally very efficient particularly when markup density is high. However, for some conditions that occur rarely and/or require complex combinations of upshifting and logic, it may be better to define simpler error-check streams that require limited postprocessing using byte matching techniques.

The techniques have been implemented and assessed for present-day commodity processors employing current SIMD technology. As processor advances see improved instruction sets and increases in width of SIMD registers, the relative advantages of the techniques over traditional byte-at-a-time sequential parsing methods is likely to increase substantially. Of particular benefit to this method, instruction set modifications that provide for more convenient carry propagation for long bitstream arithmetic would be most welcome.

A significant challenge to the application of these techniques is the difficulty of programming. The method of prototyping on unbounded bitstreams has proven to be of significant value in our work. Using the prototyping language as input to a bitstream compiler has also proven effective in generating high-performance code. Nevertheless, direct programming with bitstreams is still a specialized skill; our future research includes developing yet higher level tools to generate efficient bitstream implementations from grammars, regular expressions and other text processing formalisms.

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 2006)
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0, 5th edn. W3C Recommendation (2008)
3. Cameron, R.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)
4. Cameron, R.D., Herdy, K.S., Lin, D.: High performance XML parsing using parallel bit stream technology. In: CASCON 2008: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research, pp. 222–235. ACM Press, New York (2008)
5. Dai, Z., Ni, N., Zhu, J.: A 1 cycle-per-byte XML parsing accelerator. In: FPGA 2010: Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 199–208. ACM Press, New York (2010)
6. Herdy, K.S., Burggraf, D.S., Cameron, R.D.: High performance GML to SVG transformation for the visual presentation of geographic data in web-based mapping systems. In: Proceedings of SVG Open 2008 (August 2008)
7. Kostoulas, M.G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., 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 2006), pp. 93–102 (2006)
8. Leventhal, M., Lemoine, E.: The XML chip at 6 years. In: International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth (August 2009)
9. Shah, B., Rao, P.R., Moon, B., Rajagopalan, M.: A data parallel algorithm for XML DOM parsing. In: Bellahsène, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 75–90. Springer, Heidelberg (2009)
10. Zhang, Y., Pan, Y., Chiu, K.: Speculative p-DFAs for parallel XML parsing. In: 2009 International Conference on High Performance Computing (HiPC), pp. 388–397 (December 2009)