

# Boosting the Efficiency of Text Processing on Commodity Processors: The Parabix Story

Paper ID 160

## Abstract

In modern applications text files are employed widely. For example, XML files provide data storage in human readable format and are ubiquitous in applications ranging from database systems to mobile phone SDKs. Traditional text processing tools are built around a byte-at-a-time processing model where each character token of a document is examined. The byte-at-a-time model is highly challenging for commodity processors. It includes many unpredictable input-dependent branches which cause pipeline squashes and stalls. Furthermore, typical text processing tools perform few operations per character and experience high cache miss rates. Overall, parsing text in important domains like XML processing requires high performance motivating the adoption of custom hardware solutions.

In this paper, we enable text processing applications to effectively use commodity processors. We introduce Parabix (Parallel Bitstream) technology, a software toolchain and execution framework that allows applications to exploit modern SIMD instructions for high performance text processing. Parabix enables the application developer to write constructs assuming unlimited SIMD data parallelism and Parabix's bitstream translator generates code based on machine specifics (e.g., SIMD register widths). The key insight into efficient text processing in Parabix is the data organization. Parabix transposes the sequence of character bytes into sets of 8 parallel bit streams which then enables us to operate on multiple characters with bit-parallel SIMD operations. We demonstrate the features and efficiency of Parabix with an XML parsing application. We evaluate the Parabix-based parser against two widely used XML parsers, Expat and Apache's Xerces. Parabix makes efficient use of intra-core SIMD hardware and demonstrates  $2\times-7\times$  speedup and  $4\times$  improvement in energy efficiency compared to the conventional parsers. We assess the scalability of SIMD implementations across three generations of x86 processors including the new Sandy-Bridge. We compare the 256-bit AVX technology in Intel SandyBridge versus the now legacy 128-bit SSE technology and analyze the benefits and challenges of using the AVX extensions. Finally, we partition the XML program into pipeline stages and demonstrate that thread-level parallelism enables the application to exploits SIMD units scattered across the different cores and improves performance ( $2\times$  on 4 cores) at same energy levels as the single-thread version for the XML application.

## 1 Introduction

Classical Dennard voltage scaling enabled us to keep all of transistors afforded by Moore's law active. Dennard scaling reached its limits in 2005 and this has resulted in a rethink of the way general-purpose processors are built: frequencies have remained stagnant over the last 5 years with the capability to boost a core's frequency only if other cores on the chip are shut off. Chip makers strive to achieve energy efficient computing by operating at more optimal core frequencies and aim to increase performance with

0056 a larger number of cores. Unfortunately, given the limited levels of parallelism that can be found in  
0057 applications [4], it is not certain how many cores can be productively used in scaling our chips [11].  
0058  
0059 This is because exploiting parallelism across multiple cores tends to require heavy weight threads that are  
0060 difficult to manage and synchronize.  
0061  
0062

0064 The desire to improve the overall efficiency of computing is pushing designers to explore customized  
0065 hardware [13, 19] that accelerate specific parts of an application while reducing the overheads present  
0066 in general-purpose processors. They seek to exploit the transistor bounty to provision many different  
0067 accelerators and keep only the accelerators needed for an application active while switching off others on  
0068 the chip to save power consumption. While promising, given the fast evolution of languages and software,  
0069 its hard to define a set of fixed-function hardware for commodity processors. Furthermore, the toolchain to  
0070 create such customized hardware is itself a hard research challenge. We believe that software, applications,  
0071 and runtime models themselves can be refactored to significantly improve the overall computing efficiency  
0072 of commodity processors.  
0073  
0074

0082 In this paper, we tackle the infamous “thirteenth dwarf” (parsers/finite state machines) that is considered  
0083 to be the hardest application class to parallelize [1]. We present Parabix, a novel execution framework and  
0084 software run-time environment that can be used to dramatically improve the efficiency of text processing  
0085 and parsing on commodity processors. Parabix transposes byte-oriented character data into parallel bit  
0086 streams for the individual bits of each character byte and then exploits the SIMD extensions on commod-  
0087 ity processors (SSE/AVX on x86, Neon on ARM) to process hundreds of character positions in an input  
0088 stream simultaneously. To achieve transposition, Parabix exploits sophisticated SIMD instructions that  
0089 enable data elements to be packed and unpacked from registers in a regular manner which improves the  
0090 overall cache access behavior of the application resulting in significantly fewer misses and better utiliza-  
0091 tion. Parabix also dramatically reduces branches in parsing code resulting in a more efficient pipeline and  
0092 substantially improves register/cache utilization which minimizes energy wasted on data transfers.  
0093  
0094

0104 We apply Parabix technology to the problem of XML parsing and develop several implementations for  
0105 different computing platforms. XML is a particularly interesting application; it is a standard of the web  
0106 consortium that provides a common framework for encoding and communicating data. XML provides crit-  
0107 ical data storage for applications ranging from Office Open XML in Microsoft Office to NDFD XML of  
0108  
0109  
0110  
0111

0112 the NOAA National Weather Service, from KML in Google Earth to Castor XML in the Martian Rovers.  
0113 XML parsing efficiency is important for multiple application areas; in server workloads the key focus in  
0114 on overall transactions per second, while in applications in network switches and cell phones, latency and  
0115 energy are of paramount importance. Traditional software-based XML parsers have many inefficiencies  
0116 including considerable branch misprediction penalties due to complex input-dependent branching struc-  
0117 tures as well as poor use of memory bandwidth and data caches due to byte-at-a-time processing and  
0118 multiple buffering. XML ASIC chips have been around for over 6 years, but typically lag behind CPUs  
0119 in technology due to cost constraints. Our focus is how much we can improve performance of the XML  
0120 parser on commodity processors with Parabix technology.  
0121  
0122  
0123  
0124  
0125  
0126  
0127  
0128  
0129

0130 In the end, as summarized by Figure 1 our Parabix-based XML parser improves the performance and  
0131 energy efficiency several-fold compared to widely-used software parsers, approaching the performance of  
0132 ASIC XML parsers.<sup>1</sup>  
0133  
0134  
0135

0136 Overall we make the following contributions in this paper.  
0137

0138 1) We outline the Parabix architecture, tool chain and run-time environment and describe how it may  
0139 be used to produce efficient XML parser implementations on a variety of commodity processors. While  
0140 studied in the context of XML parsing, the Parabix framework can be widely applied to many problems in  
0141 text processing and parsing.  
0142  
0143  
0144  
0145

0146 2) We compare Parabix XML parsers against conventional parsers and assess the improvement in overall  
0147 performance and energy efficiency on each platform. We are the first to compare and contrast SSE/AVX  
0148 extensions across multiple generation of Intel processors and show that there are performance challenges  
0149 when using newer generation SIMD extensions. We compare the ARM Neon extensions against the x86  
0150 SIMD extensions and comment on the latency of SIMD operations across these architectures.  
0151  
0152  
0153  
0154  
0155

0156 3) Finally, building on the SIMD parallelism of Parabix technology, we multithread the Parabix XML  
0157 parser to to enable the different stages in the parser to exploit SIMD units across all the cores. This further  
0158 improves performance while maintaining the energy consumption constant with the sequential version.  
0159  
0160  
0161

0162 The remainder of this paper is organized as follows. Section 2 presents background material on XML  
0163 parsing and provides insight into the inefficiency of traditional parsers on mainstream processors. Sec-  
0164  
0165

---

0166 <sup>1</sup>The actual energy consumption of the XML ASIC chips is not published by the companies.  
0167

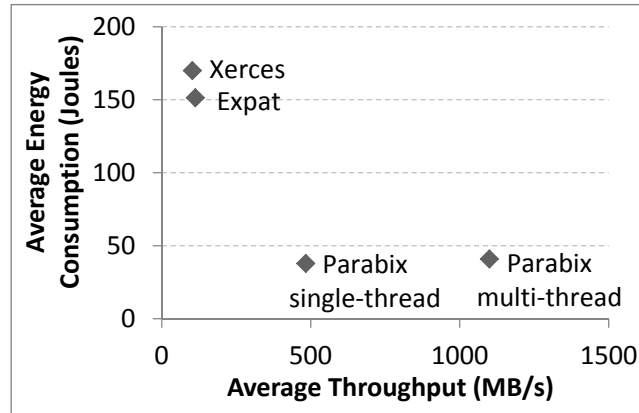


Figure 1: XML Parser Technology Energy vs. Performance

tion 3 describes the Parabix architecture, tool chain and run-time environment. Section 4 describes the application of the Parabix framework to the construction of an XML parser enforcing all the well-formedness rules of the XML specification. Section 6 presents a detailed performance analysis of Parabix on a Core-i3 system using hardware performance counters and compares it against conventional parsers. Section 7 compares the performance and energy efficiency of 128 bit SIMD extensions across three generations of intel processors and includes a comparison with the ARM Cortex-A8 processor. Section 8 examines the Intel's new 256-bit AVX technology and comments on the benefits and challenges compared to the 128-bit SSE instructions. Finally, Section 9 looks at the multithreading of the Parabix XML parser which seeks to exploit the SIMD units scattered across multiple cores.

## 2 Background

### 2.1 XML

Extensible Markup Language (XML) is a core technology standard of the World Wide Web Consortium (W3C); it provides a common framework for encoding and communicating structured and semi-structured information. XML can represent virtually any type of information (i.e., content) in a descriptive fashion. XML markup encodes a description of an XML document's storage layout and logical structure. Since XML is intended to be human-readable, markup tags are often verbose by design [5]. For example, Figure 2 provides a standard product list encapsulated within an XML document. All content is highlighted in bold. Anything that is not content is considered markup.

```
0224
0225 <Products>
0226   <Product ID="0001">
0227     <ProductName Language="English">Widget</ProductName>
0228     <ProductName Language="French">Bitoniau</ProductName>
0229     <Company>ABC</Company>
0230     <Price>$19.95</Price>
0231   </Product>
0232 </Products>
0233
0234
```

Figure 2: Sample XML Document

## 2.2 XML Parsers

Traditional XML parsers process an XML document sequentially, a single byte-at-a-time, from the first to the last character in the source text. Each character is examined to distinguish between the XML-specific markup, such as an left angle bracket ‘<’, and the content held within the document. The character that the parser is currently interpreting is commonly referred to its *cursor*. As the parser moves its cursor through the document, it alternates between markup scanning, validation, and content processing operations. In other words, traditional XML parsers operate as finite-state machines that use byte comparisons to transition between data and metadata states. Each state transition indicates the context for subsequent characters. Unfortunately, textual data tends to consist of variable-length items sequenced in generally unpredictable patterns; thus any character could be a state transition until deemed otherwise.

A major disadvantage of the sequential byte-at-a-time approach to XML parsing is that processing an XML document requires at least one conditional branch per byte of source text. For example, Xerces-C, which forms the foundation for widely deployed the Apache XML project [12], uses a series of nested switch statements and state-dependent flag tests to control the parsing logic of the program. Our analysis, which we detail in Section 6.2, found that Xerces requires between 6 - 13 branches per byte of XML to support this form of control flow, depending on the fraction of markup in the overall document. Cache utilization is also significantly reduced due to the manner in which markup and content must be scanned and buffered for future use. For instance, Xerces incurs ~100 L1 cache misses per 1000 bytes of XML. In general, while microarchitectural improvements may help the parser tide over some of these challenges (e.g., cache misses), the fundamental data and control flow in the parsers are ill suited for commodity processors and experience significant overhead.

### 3 The Parabix Framework

This section presents an overview of the SIMD-based parallel bit stream text processing framework, *Parabix*. The framework has three components: a unifying architectural view of text processing in terms of parallel bit streams, a tool chain for automating the generation of parallel bit stream code from higher-level specifications, and a run-time environment providing a portable SIMD programming abstraction, independent of the specific facilities available on particular target architectures.

#### 3.1 Parallel Bit Streams

The fundamental difference between the Parabix framework and traditional text processing models is in how Parabix represents the source data. Given a traditional byte-oriented text stream, Parabix first transposes the text data to a transform domain consisting of 8 parallel bit streams, known as *basis bit streams*. In essence, each basis bit stream  $b_k$  represents the stream of  $k$ -th bit of each byte in the source text. That is, the  $k$ -th bit of  $i$ -th byte in the source text is in the  $i$ -th (bit) position of the  $k$ -th basis bit stream,  $b_k$ . For example, in Figure 3, we show how the ASCII string “b7<A” is represented as 8 basis bit streams,  $b_{0\dots7}$ . The bits used to construct  $b_7$  have been highlighted in this example.

STRING	b	7	<	A
ASCII	0110001 <b>0</b>	0011011 <b>1</b>	0011111 <b>0</b>	0100000 <b>1</b>

---

b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>	b <sub>7</sub>
0	1	1	0	0	0	1	<b>0</b>
0	0	1	1	0	1	1	<b>1</b>
0	0	1	1	1	1	0	<b>0</b>
0	1	0	0	0	0	0	<b>1</b>

Figure 3: Example 7-bit ASCII Basis Bit Streams

The advantage of the parallel bit stream representation is that we can use the 128-bit SIMD registers commonly found on commodity processors (e.g. SSE on Intel) to process 128 byte positions at a time using bitwise logic, shifting and other operations.

Just as forward and inverse Fourier transforms are used to transform between the time and frequency domains in signal processing, bit stream transposition and inverse transposition provides “byte space” and “bit space” views of text. The goal of the Parabix framework is to support efficient text processing using

0336 these two equivalent representations in the same way that efficient signal processing benefits from the use  
0337 of the frequency domain in some cases and the time domain in others.  
0338

0339 In the Parabix framework, basis bit streams are used as the starting point to determine other bit streams.  
0340 In particular, Parabix uses the basis bit streams to construct *character-class bit streams* in which each 1 bit  
0341 indicates the presence of a significant character (or class of characters) in the parsing process. Character-  
0342 class bit streams may then be used to compute *lexical bit streams* and *error bit streams*, which Parabix  
0343 uses to process and validate the source document. The remainder of this section will discuss each type of  
0344 bit stream.  
0345

0346 **Basis Bit Streams:** To construct the basis bit streams, the source data is first loaded in sequential order  
0347 and then transposed — through a series of SIMD pack, shift, and bitwise operations — so that Parabix  
0348 can efficiently produce the character-class bit streams. Using the SIMD capabilities of current commodity  
0349 processors, the transposition process incurs an amortized cost of approximately 1 cycle per byte.  
0350

0351 **Character-class Bit Streams:** Typically, as text parsers process input data, they locate specific charac-  
0352 ters to determine if and when to transition between data and metadata parsing. For example, in XML, any  
0353 opening angle bracket character, ‘<’, may indicate that we are starting a new markup tag. Traditional byte-  
0354 at-a-time parsers find these characters by comparing the value of each byte with a set of known significant  
0355 characters and branching appropriately when one is found, typically using an if or switch statement. Using  
0356 this method to perform multiple transitions in parallel is non-trivial and may require fairly sophisticated  
0357 algorithms to do so correctly.  
0358

0359 Character-class bit streams allow us to perform up to 128 “comparisons” in parallel with a single op-  
0360 eration by using a series of boolean-logic operations<sup>2</sup> to merge multiple basis bit streams into a sin-  
0361 gle character-class stream that marks the positions of key characters with a 1. For example, a char-  
0362 acter is an ‘<’ if and only if  $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3 \wedge b_4 \wedge b_5) \wedge \neg(b_6 \vee b_7) = 1$ . Classes of charac-  
0363 ters can be found with similar formulas. For example, a character is a number [0–9] if and only if  
0364  $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3) \wedge \neg(b_4 \wedge (b_5 \vee b_6))$ . An important observation here is that a range of characters can  
0365 sometimes take fewer operations and require fewer basis bit streams to compute than individual characters.  
0366 Finding optimal solutions to all character-classes is non-trivial and goes beyond the scope of this paper.  
0367

0368 <sup>2</sup> $\wedge$ ,  $\vee$  and  $\neg$  denote the boolean AND, OR and NOT operations.  
0369  
0370  
0371  
0372  
0373  
0374  
0375  
0376  
0377  
0378  
0379  
0380  
0381  
0382  
0383  
0384  
0385  
0386  
0387  
0388  
0389

**Lexical and Error Bit Streams:** 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 integer math. 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.

To form lexical bit streams, we have to introduce a few new operations: `Advance` and `ScanThru`. 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 4 suppose we have the regular expression  $\langle [a-zA-Z]^+ \rangle$  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 " $\langle \rangle$ " 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 " $\langle \text{error} \rangle$ " too fails to match the expected pattern. With additional post bit-stream processing, the erroneous cursors in  $L_0$  and  $L_1$  can be removed; the details of which go beyond the scope of this paper.

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

Figure 4: Lexical Parsing in Parabix



0448 Using this parallel bit stream approach, conditional branch statements used to identify key positions  
0449 and/or syntax errors at each each parsing position are mostly eliminated, which, as Section 6.2 shows,  
0450 minimizes branch misprediction penalties. Accurate parsing and parallel lexical analysis is done through  
0451 processor-friendly equations and requires neither speculation nor multithreading.  
0452  
0453  
0454  
0455

## 0456 3.2 Parabix Compilers

0457 To support the Parabix execution framework, we have developed a tool chain to automate various  
0458 aspects of parallel bit stream programming. Our tool chain consists of two compilers: a character class  
0459 compiler (*ccc*) and an unbounded bit stream to C/C++ block-at-a-time processing compiler (*Pablo*).  
0460  
0461  
0462  
0463  
0464

0465 The character class compiler is used to automatically produce bit stream logic for all the individual  
0466 characters (e.g., delimiters) and character classes (e.g., digits, letters) used in a particular application.  
0467 Input is specified using a character class syntax adapted from the standard regular expression notations.  
0468  
0469 Output is a minimized set of three-address bitwise operations to compute each of the character classes  
0470 from the basis bit streams.  
0471  
0472  
0473  
0474

0475 For example, Figure 5 shows the input and output produced by the character class compiler for the  
0476 example of [0-9] discussed in the previous section. The output operations may be viewed as operations  
0477 on a single block of input at a time, or may be viewed as operations on unbounded bit streams as supported  
0478  
0479  
0480  
0481 by the Pablo compiler.  
0482

```
0483 INPUT: digit = [0-9]  
0484  
0485  
0486 OUTPUT: temp1 = (basis_bits.bit_0 | basis_bits.bit_1)  
0487 temp2 = (basis_bits.bit_2 & basis_bits.bit_3)  
0488 temp3 = (temp2 & ~ temp1)  
0489 temp4 = (basis_bits.bit_5 | basis_bits.bit_6)  
0490 temp5 = (basis_bits.bit_4 & temp4)  
0491 digit = (temp3 & ~ temp5)  
0492  
0493
```

0494 Figure 5: Character Class Compiler Input/Output

0495  
0496  
0497 The Pablo compiler abstracts away the details of programming parallel bit stream code in terms of finite  
0498 SIMD register widths and application buffer sizes. Input to Pablo is a language for expressing bit stream  
0499 operations on unbounded bit streams. The operations include bitwise logic, the `Advance` and `ScanThru`  
0500  
0501  
0502  
0503

```

0504     INPUT: def parse_tags(classes, errors):
0505                 classes.C0 = Alpha
0506                 classes.C1 = Rangle
0507                 classes.C2 = Langle
0508                 L0 = bitutil.Advance(C2)
0509                 errors.E0 = L0 &~ C0
0510                 L1 = bitutil.ScanThru(L0, C0)
0511                 errors.E1 = L1 &~ C1
0512
0513
0514
0515     OUTPUT: struct Parse_tags {
0516                 Parse_tags() { CarryInit(carryQ, 2); }
0517                 void do_block(Classes & classes, Errors & errors) {
0518                     BitBlock L0, L1;
0519                     classes.C0 = Alpha;
0520                     classes.C1 = Rangle;
0521                     classes.C2 = Langle;
0522                     L0 = BitBlock_advance_ci_co(C2, carryQ, 0);
0523                     errors.E0 = simd_andc(L0, C0);
0524                     L1 = BitBlock_scanthru_ci_co(L0, C0, carryQ, 1);
0525                     errors.E1 = simd_andc(L1, C1);
0526                     CarryQ_Adjust(carryQ, 2);
0527                 }
0528                 CarryDeclare(carryQ, 2);
0529             };
0530
0531
0532
0533
0534
0535
0536
0537
0538
0539
0540
0541
0542
0543
0544
0545
0546
0547
0548
0549
0550
0551
0552
0553
0554
0555
0556
0557
0558
0559

```

Figure 6: Parallel Block Compiler (Pablo) Input/Output

operations described in the previous subsection as well as if and while control structures. Pablo translates these operations to block-at-a-time code in C/C++.

The key functionality of Pablo is to arrange for block-to-block carry bit propagation to implement the long bit stream shift and addition operations required by `Advance` and `ScanThru`.

For example, we can translate the simple parsing example of 4 above into Pablo code to produce the output as shown in Figure 6. In this example, Pablo has the primary responsibility of inserting carry variable declarations that allow the results of `Advance` and `ScanThru` operations to be carried over from block to block. A separate carry variable is required for every `Advance` or `ScanThru` operation. A function containing such operations is translated into a public C++ class (struct), which includes a Carry Queue to hold all the carry variables from iteration to iteration, together with the a method `do_block` to implement the processing for a single block (based on the SIMD register width). Macros `CarryDeclare`

0560 and `CarryInit` declare and initialize the Carry Queue structure depending on the specific architecture  
0561 and Carry Queue representation. The unbounded bit stream `Advance` and `ScanThru` operations are  
0562 translated into block-by-block equivalents with explicit carry-in and carry-out processing. At the end of  
0563 each block, the `CarryQ_Adjust` operation implements any necessary adjustment of the Carry Queue to  
0564 prepare for the next iteration. The Pablo language and compiler also support conditional and iterative bit  
0565 stream logic on unbounded streams (if and while constructs) which involves additional carry-test insertion  
0566 in control branches. Explaining the full details of the translation is beyond the scope of this paper.  
0567  
0568  
0569  
0570  
0571  
0572  
0573

### 0574 3.3 Parabix Run-Time Libraries

0575 The Parabix architecture also includes run-time libraries that support a machine-independent view of  
0576 basic SIMD operations, as well as a set of core function libraries. For machine-independence, we program  
0577 all operations using an abstract SIMD machine. The abstract machine supports all power-of-2 field widths  
0578 up to the full SIMD register width on a target machine. Let  $w = 2k$  be the field width in bits. Let  $f$  be a  
0579 basic binary operation defined on  $w$ -bit quantities producing an  $w$ -bit result. Let  $W$  be the SIMD vector  
0580 size in bits where  $W = 2K$ . Then the C++ template notation `v=simd<w>::f(a,b)` denotes the general  
0581 pattern for a vertical SIMD operation yielding an output SIMD vector  $v$ , given two input SIMD vectors  $a$   
0582 and  $b$ . For each field  $v_i$  of  $v$ , the value computed is  $f(a_i, b_i)$ . For example, given 128-bit SIMD vectors,  
0583 `simd<8>::add(a,b)` represents the simultaneous addition of sixteen 8-bit fields.  
0584  
0585  
0586  
0587  
0588  
0589  
0590  
0591  
0592  
0593  
0594

0595 These operations were originally developed for 128-bit AltiVec operations on Power PC as well as 64-  
0596 bit MMX and 128-bit SSE operations on Intel but have recently extended to support the new 256-bit AVX  
0597 operations on Intel as well as the 128-bit Neon operations on the ARM architecture.  
0598  
0599  
0600

## 0601 4 The Parabix XML Parser

0602 This section describes the implementation of the Parabix XML parser. Figure 7 shows its overall  
0603 structure set up for well-formedness checking. The input file is processed using 11 functions organized  
0604 into 7 modules. In the first module, `Read_Data`, the input file is loaded into the `data_buffer`. The  
0605 data is then transposed to eight parallel basis bit streams (`basis_bits`) in the `Transposition` module.  
0606 The `basis_bits` are used in by the `U8_Validation` module to validate UTF-8 characters, and by the  
0607 `Classification` and `Gen_Scope` module to generate all the XML lexical item streams (`lex`) and  
0608  
0609  
0610  
0611  
0612  
0613  
0614  
0615

0616  
0617  
0618  
0619  
0620  
0621  
0622  
0623  
0624  
0625  
0626  
0627  
0628  
0629  
0630  
0631  
0632  
0633  
0634  
0635  
0636  
0637  
0638  
0639  
0640  
0641  
0642  
0643  
0644  
0645  
0646  
0647  
0648  
0649  
0650  
0651  
0652  
0653  
0654  
0655  
0656  
0657  
0658  
0659  
0660  
0661  
0662  
0663  
0664  
0665  
0666  
0667  
0668  
0669  
0670  
0671

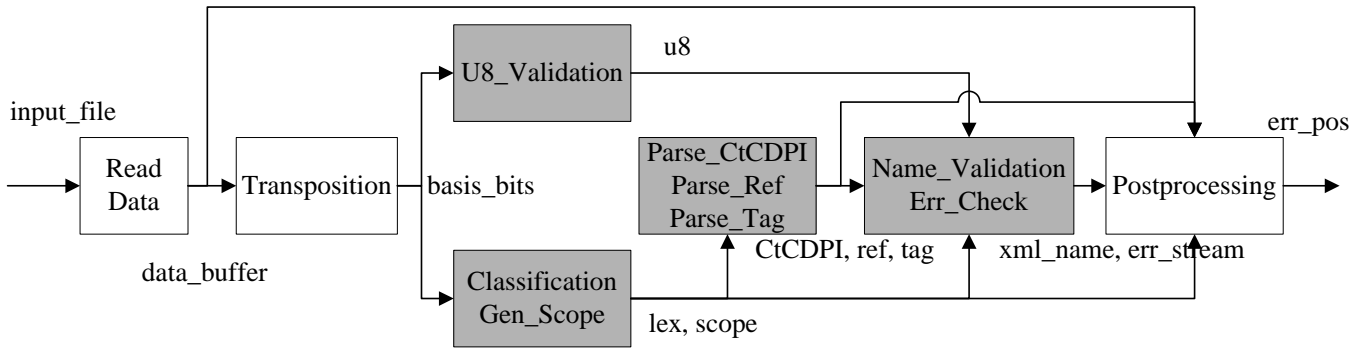


Figure 7: Parabix XML Parser Structure

scope streams (scope). Scope streams are a simplified subset of lex streams in which the legal yet insignificant cursors have been removed. Both the lex and scope streams are supplied to the parsing module, which consists of three functions: (1) *Parse\_CtCDPI*, (2) *Parse\_Ref* and (3) *Parse\_tag*; these functions deal with the parsing of (1) comments, CDATA sections, and processing instructions; (2) references, and (3) start tags, end tags, and empty tags as well as any related attributes. Afterwards, the information is gathered by the *Name\_Validation* and *Err\_Check* functions, producing name check streams and error streams. Name check streams are weak error streams that verify each character used in a name is valid according to the XML 1.0 specification. These are then passed to the final *Postprocessing* module. Any error that cannot be conveniently detected in bit space are checked here. The final output reports any well-formedness error and its position within the input file.

Using this structure, all of the functions in the four shaded modules consist entirely of parallel bit stream operations. Of these, the *Classification* function consists of XML character class definitions that are generated using our character class compiler *ccc*, while much of the *U8\_Validation* similarly consists of UTF-8 byte class definitions that are also generated by *ccc*. The remainder of these functions are programmed using our unbounded bit stream language following the logical requirements of XML parsing. All the functions in the four shaded modules are then compiled to low-level C/C++ code using our Pablo compiler. This code is then linked in with the general *Transposition* code available in the Parabix run-time library, as well as the hand-written *Postprocessing* code that completes the well-formed checking.

## 5 Evaluation Framework

File Name	dew.xml	jaw.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
Markup Density	0.07	0.13	0.57	0.76	0.87

Table 1: XML Document Characteristics

**XML Parsers** We evaluate the Parabix XML parser described above against two widely available open-source parsers, Xerces-C++, and Expat. Each of the parsers is evaluated on the task of implementing the parsing and well-formedness checking requirements of the full XML 1.0 specification [5]. Xerces-C++ version 3.1.1 (SAX) [12] is a validating open source XML parser written in C++ available as part of the the Apache project. To ensure a fair comparison, we use the WFXML scanner of Xerces to eliminate the overheads of validation and also use the SAX interface to avoid the overheads costs of DOM tree construction. Expat version 2.0.1 [8] is a non-validating XML parser library written in C.

**XML Workloads** XML is used for a variety of purposes ranging from databases to config files in mobile phones. A key feature of these XML files that affects the overall parsing performance is the *Markup density*. *Markup density* is defined as the ratio of the total markup contained within an XML file to the total XML document size. This metric has substantial influence on the performance of traditional recursive descent XML parser implementations. We use a mixture of document-oriented and data-oriented XML files in our study to analyze workloads with a full spectrum of markup densities.

Table 1 shows the document characteristics of the XML input files selected for this performance study. The jawiki.xml and dewiki.xml XML files represent document-oriented XML inputs and contain the three-byte and four-byte UTF-8 sequence required for the UTF-8 encoding of Japanese and German characters respectively. The remaining data files are data-oriented XML documents and consist entirely of single byte encoded ASCII characters.

**Platform Hardware** SSE extensions have been available on commodity Intel processors for over a decade since the Pentium III. They have steadily evolved with improvements in instruction latency, cache interface, and register resources, and the addition domain specific instructions. Here we investigate SIMD extensions across three different generations of intel processors. Table 2 describes the Intel multicores we investigate. We compare the energy and performance profile of the Parabix under the platforms. We also

analyze the implementation specifics of SIMD extensions under various microarchitecture. We we evaluate both the legacy SSE and newer AVX extensions supported by Sandybridge.

We propose to investigate each the execution profiles of XML parsers using the the Performance Monitoring Counter (PMC) hardware event found in the processor. We have chosen several key hardware performance events which provide insight into the profile of our application and indicate if the processor is doing useful work [2, 3]. The set of performance counters included in our study are Branch instructions, Branch mispredictions, Integer instructions, SIMD instructions, and Cache misses. In addition, we characterize the SIMD operations and study the type and class of SIMD operations using the Intel Pin binary instrumentation framework.

Processor	Core2 Duo (2.13GHz)	i3-530 (2.93GHz)	Sandybridge (2.80GHz)
L1 D Cache	32KB	32KB	32KB
L2 Cache	Shared 2MB	256KB/core	256KB/core
L3 Cache	—	4MB	6MB
Bus or QPI	1066Mhz Bus	1333Mhz QPI	1333Mhz QPI
Memory	2GB	4GB	6GB
Max TDP	65W	73W	95W

Table 2: Platform Hardware Specs

**Energy Measurement** A key benefit of the Parabix parser is its more efficient use of the processor pipeline which reflects in the overall energy usage. We measure the energy consumption of the processor directly using a current clamp. We apply the Fluke i410 current clamp [9] to the 12V wires that supply power to the processor sockets. The clamp detects the magnetic field created by the flowing current and converts it into voltage levels (1mV per 1A current). The voltage levels are then monitored by an Agilent 34410a digital multimeter at the granularity of 100 samples per second. This measurement captures the instantaneous power to the processor package, including cores, caches, northbridge memory controller, and the quick-path interconnects. We obtain samples throughout the entire execution of the program and then calculate overall total energy as  $12V * \sigma_{i=1}^{N_{samples}} Sample_i$ .

## 6 Efficiency of the Parabix

In this section we analyze the energy and performance characteristics of the Parabix-based XML parser against the software XML parsers, Xerces and Expat. For our baseline evaluation, we compare all the

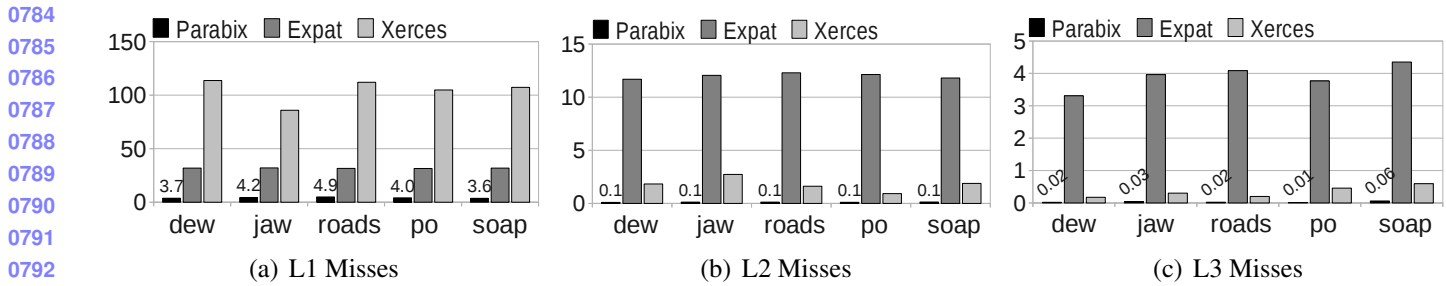


Figure 8: Cache Misses per kB of input data.

XML parsers on a fixed platform, the Core-i3.

## 6.1 Cache behavior

The approximate miss penalty in Core-i3 for L1, L2 and L3 caches is 4, 11, and 36 cycles respectively. The L1 (32KB) and L2 cache (256KB) are private per core, while the 4MB L3 is shared by all the cores. Figure 8 shows the cache misses per kilobyte of input data. Analytically, the cache misses for the Expat and Xerces parsers represent a 0.5 cycle per XML byte processed. This overhead does not necessarily reflect in the overall performance of these parsers as they experience other overheads related to branch mispredictions. Parabix’s data reorganization significantly improves the overall cache miss rate. We experience  $7\times$  less misses than Expat and  $25\times$  less misses than Xerces at the L1 and  $104\times$  less misses than Expat and  $15\times$  less misses than Xerces at the L2 level. The improved cache utilization keeps the SIMD units busy and prevent memory related stalls. Note that cache misses also cause increased application energy consumption due to increased energy required to access higher levels in the cache hierarchy. We estimated with microbenchmarks that the L1, L2, and L3 cache misses consume approximately 8.3nJ, 19nJ, and 40nJ respectively. For a 1GB XML file Expat and Xerces would consume over 0.6J and 0.9J respectively due to cache misses alone.

## 6.2 Branch Mispredictions

In general, reducing the branch misprediction rate is difficult in text-based XML parsing applications. This is due to (1) variable length nature of the syntactic elements contained within XML documents, (2) a data dependent characteristic, and (3) the extensive set of syntax constraints imposed by the XML. Traditional byte-at-a-time XML parser’s performance is limited by the number of branch mispredictions. As shown in Figure 9(a), Xerces averages up to 13 branches per XML byte processed on high density

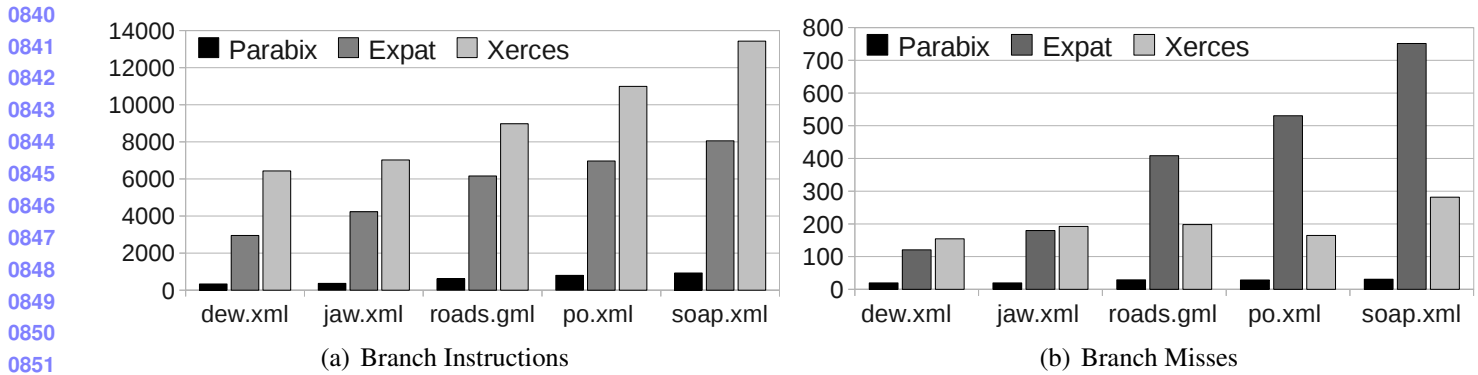


Figure 9: Branch characteristics on the Core-i3 per kB of input data.

markup. On modern commodity processors the cost of a single branch misprediction is incur over 10s of CPU cycles to restart the processor pipeline. The high miss prediction rate in conventional parsers add significant overhead. In Parabix the transformation to SIMD operation eliminates many branches. Further optimizations take advantage of Parabix’s data organization and replace condition branches with *bit scan* operations that can process up to 128 characters worth of branches with one operation. In many cases, we also replace the branches with logical predicate operations. Our predicates are cheaper to compute since they involve only bit parallel SIMD operations.

As shown in Figure 9(a), Parabix processing is almost branch free. Parabix exhibits minimal dependence on source XML markup density; it experiences between 19.5 and 30.7 branch mispredictions per thousand of XML byte. The cost of branch mispredictions for the Expat parser can be over 7 cycles per XML byte (see Figure 9(b)) —this cost alone is higher than the average latency of a byte processed by Parabix.

### 6.3 SIMD Instructions vs. Total Instructions

In Parabix, bit streams are both computed and predominately operated upon using the SIMD instructions of commodity processors. The ratio of retired SIMD instructions to total instructions provides insight into the relative degree to which Parabix achieves parallelism over the byte-at-a-time approach.

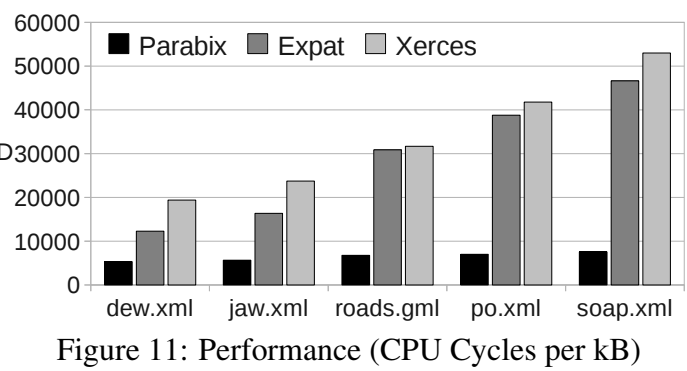
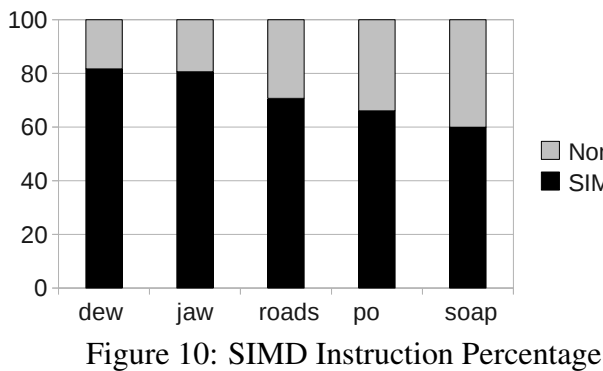
Using the Intel Pin tool, we gather the dynamic instruction mix for each XML workload, and classify instructions as either vector (SIMD) or non-vector instructions. Figure 10 shows the percentage of SIMD instructions for the Parabix XML parser. The ratio of executed SIMD instructions over total instructions indicates the amount of parallel processing we were able to extract. The Parabix instruction mix is made up of 60% to 80% SIMD instructions. The markup density of the files influence the number of scalar



instructions needed to handle the tag processing which affects the overall parallelism that can be extracted by Parabix. We find that degradation rate is low and thus the performance penalty incurred by increasing the markup density is minimal.

### 6.4 CPU Cycles

Figure 11 shows overall parser performance evaluated in terms of CPU cycles per kilobyte. The Parabix parser is 2.5x to 4x faster on document-oriented input and 4.5 to 7 times faster on data-oriented input. Traditional parsers can be dramatically slowed by dense markup, while Parabix is affected much less. The results presented are not entirely fair to the Xerces parser since it first transcodes input from UTF-8 to UTF-16 before processing. In Xerces, this transcoding requires several cycles per byte. However, transcoding using parallel bit streams is significantly faster and requires less than a single cycle per byte.



### 6.5 Power and Energy

In this section, we study the power and energy consumption of Parabix in comparison with Expat and Xerces on Core-i3. The average power of Core-i3 is about 21 watts. Figure 12(a) shows the average power consumed by each parser. Parabix, dominated by SIMD instructions which uses approximately 5% additional power. While the SIMD functional units are significantly wider than the scalar counterparts; register width and functional unit power account only for a small fraction of the overall power consumption in a processor pipeline. More importantly by using data parallel operations Parabix amortizes the fetch and data access overheads. This results in minimal power increase compared to the conventional parsers. Perhaps the energy trends shown in Figure 12(b) reveal an interesting trend. Parabix consumes substantially less energy than the other parsers. Parabix consumes 50 to 75 nJ per byte while Expat and Xerces consume

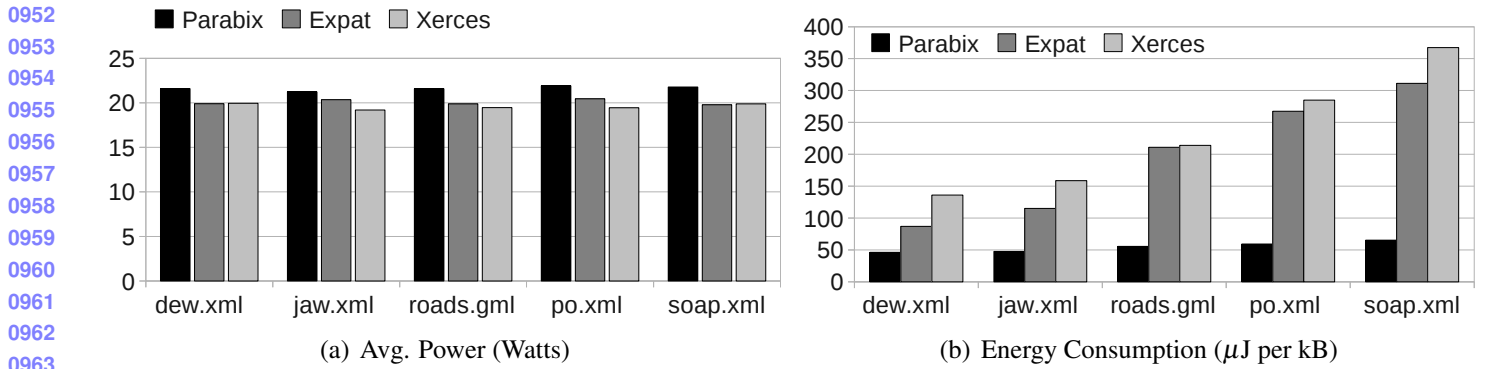


Figure 12: Power profile of Parabix on Core-i3

80nJ to 320nJ and 140nJ to 370nJ per byte respectively. Although Parabix requires slightly more power (per instruction), the processing time of Parabix is significantly lower.

## 7 Parabix on various hardware

### 7.1 Performance

In this section, we study the performance of the XML parsers across three generations of Intel architectures. Figure 13 (a) shows the average execution time of Parabix. We analyze the execution time in terms of SIMD operations that operate on bitstreams (*bit-space*) and scalar operations that perform post processing on the original character bytes. In Parabix a significant fraction of the overall execution time is spent in SIMD operations.

Our results demonstrate that Parabix’s optimizations are complementary to hardware improvements and seem to further improve the efficiency of newer microarchitectures. For Parabix’s bit-stream processing, Core-i3 results in an 40% performance improvement over Core2, whereas SandyBridge results in a 20% improvement compared to Core-i3. The improvements in the bit-space SIMD operations is stable across the different input files. Postprocessing operations demonstrate data dependent variance. Core-i3 gains between 27%—40% compared to Core2 and SandyBridge gains between 16%—39% compared to Core-i3. For the purpose of comparison, Figure 13 (b) shows the performance of the Expat parser; Core-i3 improves performance only by 5% over Core2 while SandyBridge improves performance by less than 10% over Core-i3. Not that the gains of Core-i3 over Core2 includes an improvement both in the clock frequency and microarchitecture improvements while SandyBridge’s gains can be mainly attributed to the

1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063

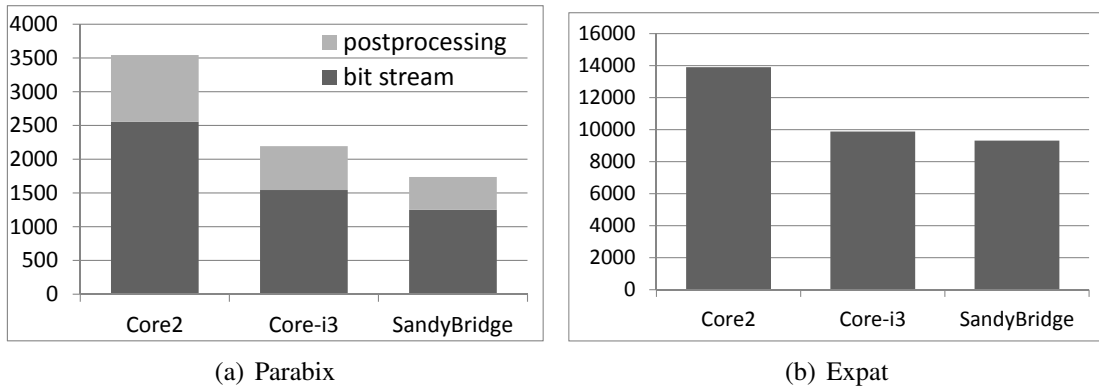


Figure 13: Average Performance Parabix vs. Expat (y-axis: ns per kB)

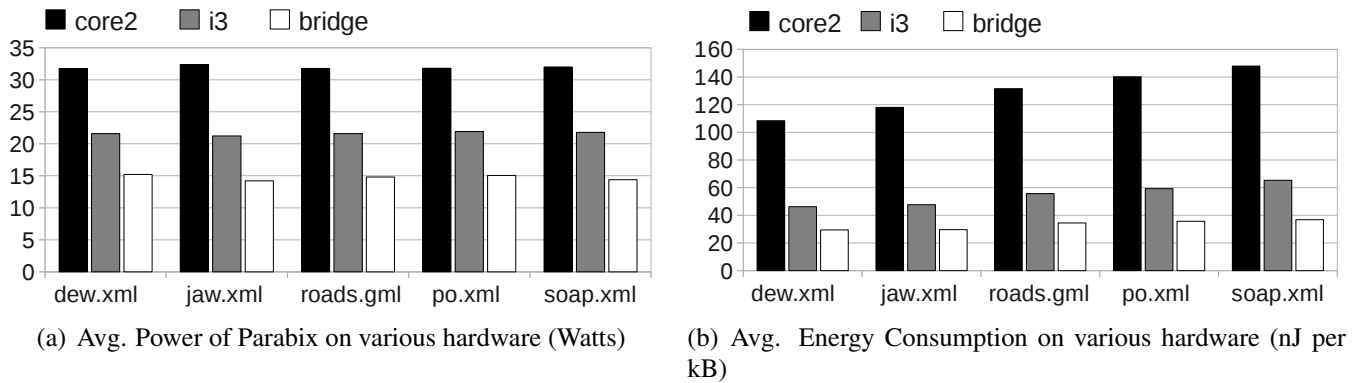


Figure 14: Energy Profile of Parabix on various hardware platforms

architecture.

Figure 14(a) shows the average power consumption of Parabix over each workload and as executed on each of the processor cores — Core2, Core-i3 and SandyBridge. Overall the last three generation of processors seem to bring with them 25—30% improvement in power consumption with every generation. Parabix on SandyBridge consumes less than 15W. Overall, Parabix on SandyBridge consumes 72% to 75% less energy than Core2.

## 7.2 Parabix on Mobile processors

Our experience with the generation of Intel processors led us to contemplate about mobile processors such as the ARM Cortex-A8 which also includes SIMD units. ARM NEON makes available a 128-bit SIMD instruction set similar in functionality to Intel SSE3 instruction set. In this section, we present our performance comparison of a NEON-based port of Parabix versus the Expat parser. Xerces is excluded from this portion of our study due to the complexity of the cross-platform build process for C++

1064 applications.

1065 The platform we use is the Samsung Galaxy Android Tablet that houses a Samsung S5PC110 ARM  
1066 Cortex-A8 1Ghz single-core, dual-issue, superscalar microprocessor. It includes a 32kB L1 data cache  
1067 and a 512kB L2 shared cache. Migration of Parabix to the Android platform began with the retargeting of  
1068 a subset of the Parabix SIMD library for ARM NEON. The majority of the Parabix SIMD functionality  
1069 ported directly. However, for a small subset of the SIMD functions (e.g., bit packing) of NEON equivalents  
1070 did not exist. In such cases we simply emulated logical equivalent instructions using the available the  
1071 scalar instruction set. This library code was cross-compiled for Android using the Android NDK.  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079

1080 A comparison of Figure 15(a) and Figure 11 demonstrates that the performance of both Parabix and  
1081 Expat degrades substantially on Cortex-A8 ( $5\times$ — $17\times$ ). This result was expected given the comparably  
1082 performance limited Cortex-A8. Surprisingly, on Cortex-A8, Expat outperforms Parabix on each of the  
1083 lower markup density workloads, dew.xml and jaw.xml. On the remaining higher-density workloads,  
1084 Parabix performs only moderately better than Expat. Investigating causes for this performance degradation  
1085 for Parabix led us to investigate the latency of Neon SIMD operations.  
1086  
1087  
1088  
1089  
1090  
1091

1092 Figure 15(b) investigates the performance of Expat and Parabix for the various input workloads on the  
1093 Cortex-A8; Figure 15(c) plots the performance for Core-i3. The results demonstrate that that the execu-  
1094 tion time of each parser varies in a linear fashion with respect to the markup density of the file. On the  
1095 both Cortex-A8 and Core-i3 both parsers demonstrate the same trend. For lower mark up density files for  
1096 which the fraction of SIMD operations and hence the potential for parallelism is limited, the overheads of  
1097 SIMD instructions affect overall execution time. Figure 15(b) provides insight into the problem, Parabix's  
1098 performance is hindered by SIMD instruction latency for low markup density files; it appears that the  
1099 latency of SIMD operations is relatively higher on the Cortex-A8 processor. This is possibly because the  
1100 Neon SIMD extensions are implemented as a coprocessor on Cortex-A8 which imposes higher overhead  
1101 for applications that frequently inter-operate between scalar and SIMD registers. Future performance en-  
1102 hancement to ARM NEON that implement the Neon within the core microarchitecture could substantially  
1103 improve the efficiency of Parabix.  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119

1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175

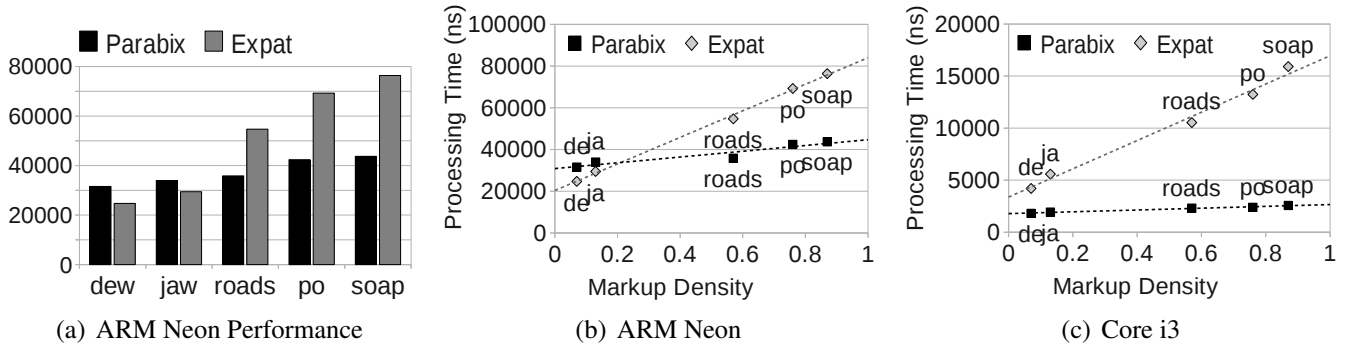


Figure 15: Comparing Parabix on ARM and Intel.

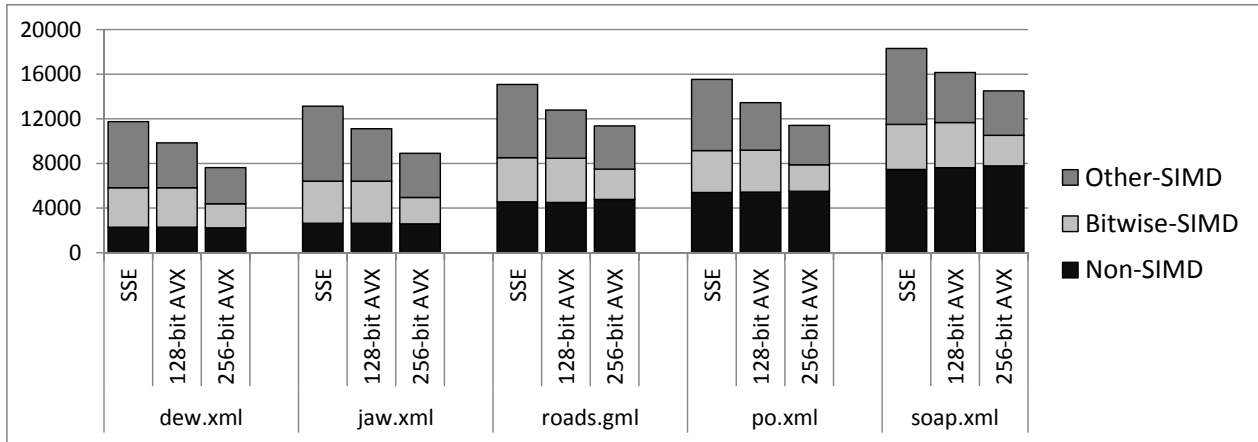


Figure 16: Parabix Instruction Counts (y-axis: Instructions per kB)

## 8 Scaling Parabix for AVX

In this section, we discuss the scalability and performance advantages of our 256-bit AVX (Advanced Vector Extensions) Parabix XML port. The Parabix SIMD libraries originally targeted the 128-bit SSE2 SIMD technology available on all modern 64-bit Intel and AMD processors but has recently been ported to AVX. AVX technology is commercially available on the latest the SandyBridge microarchitecture Intel processors. While we have to port our runtime framework the application didn't need to be modified.

**3-Operand Form** In addition to the widening of 128-bit operations to 256-bit operations, AVX technology uses a nondestructive 3-operand instruction format. Previous SSE implementations used a destructive 2-operand instruction format. In the 2-operand format a single register is used as both a source and destination register. For example,  $a = a [op] b$ . As such, 2-operand instructions that require the value of both  $a$  and  $b$ , must either copy an additional register value beforehand, or reconstitute or reload a register

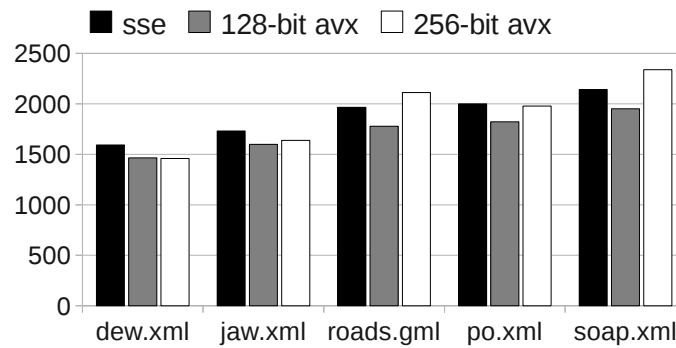


Figure 17: Parabix Performance (y-axis: ns per kB)

value afterwards to recover the value. With the 3-operand format, output may now be directed to the third register independently of the source operands. For example,  $c = a [\text{op}] b$ . By avoiding the copying or reconstituting of operand values, a considerable reduction in instructions required for unloading from and loading into registers. AVX technology makes available the 3-operand form for both the new 256-bit operations as well as the base 128-bit SSE operations.

## 8.1 256-bit Operations

With the introduction of 256-bit SIMD registers, and under ideal conditions, one would anticipate a corresponding 50% reduction in the SIMD instruction count of Parabix on AVX. However, in the Sandy-Bridge AVX implementation, Intel has focused primarily on floating point operations as opposed to the integer based operations. 256-bit SIMD is available for loads, stores, bitwise logic and floating operations, whereas SIMD integer operations and shifts are only available in the 128-bit form.

## 8.2 Performance Results

We implemented two versions of Parabix using AVX technology. The first was simply the recompilation of the existing Parabix source code written to take advantage of the 3-operand form of AVX instructions while retaining a uniform 128-bit SIMD processing width. The second involved rewriting the internal library functions of Parabix to leverage the 256-bit AVX operations wherever possible and to simulate the remaining operations using pairs of 128-bit operations. Figure 16 shows the reduction in instruction counts achieved in these two versions. For each workload, the base instruction count of the Parabix binary compiled in SSE-only mode is indicated by “sse,” the version which only takes advantage of the AVX 3-operand mode is labeled “128-bit avx,” and the version reimplemented to use 256-bit operations wherever

1232 possible is labelled “256-bit avx.” The instruction counts are divided into three classes: “non-SIMD”  
1233 operations are the general purpose instructions. The “bitwise SIMD” class comprises the bitwise logic  
1234 operations, that are available in both 128-bit form and 256-bit form. The “other SIMD” class comprises  
1235 all other SIMD operations, primarily comprising the integer SIMD operations that are available only at  
1236 128-bit widths even under AVX.  
1237  
1238  
1239

1240 Note that, in each workload, the number of non-SIMD instructions remains relatively constant with each  
1241 workload. As may be expected the number of *bit-parallel SIMD* operations remains the same for both SSE  
1242 and 128-bit while dropping dramatically when operating 256-bits at a time. The reduction measured was  
1243 32%–39% depending on workload because some bitwise logic needed in implementation is composed of  
1244 128-bit operations. This limits the performance gains achieved when using the AVX instructions. The  
1245 “other SIMD” class shows a substantial 30%–35% reduction with AVX 128-bit technology compared to  
1246 SSE. This reduction is due to elimination of register unloading and reloading when SIMD operations  
1247 are compiled using 3-operand AVX form versus 2-operand SSE form. A further 10%–20% reduction is  
1248 observed with Parabix version rewritten to use 256-bit operations.  
1249

1250 The reductions in instruction counts are quite dramatic with the AVX extensions in Parabix demonstrat-  
1251 ing the ability of our runtime framework to exploit the available hardware resources. As shown in Figure  
1252 **17**, the benefits of the reduced SIMD instruction count are achieved only in the AVX 128-bit version. In  
1253 this case, the benefits of 3-operand form seem to fully translate to performance benefits. Based on the  
1254 reduction of overall Bitwise-SIMD instructions we expected a 11% improvement in performance. Instead,  
1255 perhaps bizarrely, the performance of Parabix in the 256-bit AVX implementation does not improve sig-  
1256 nificantly and actually degrades for files with higher markup density (average 11%). *dew.xml*, on which  
1257 bitwise-SIMD instructions reduced by 39%, saw a performance improvement of 8%. We believe that  
1258 this is primarily due to the intricacies of the first generation AVX implementation in SandyBridge, with  
1259 significant latency in many of the 256-bit instructions in comparison to their 128-bit counterparts. The  
1260 256-bit instructions also have different scheduling constraints that seem to reduce overall throughput. If  
1261 these latency issues can be addressed in future AVX implementations, further substantial performance and  
1262 energy benefits could be realized in XML parsing with Parabix.  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287

## 9 Multithreaded Parabix

The general challenge with boosting performance through multicore parallelism is the power and energy overheads. When we put more cores to work on a chip it results in proportionate increase in power. Unfortunately, due to the runtime overheads associated with thread management and data synchronization it is very hard to obtain corresponding improvements in performance resulting in increased energy costs. Parabix, which exploits intra-core fine-grain SIMD-based can improve performance and achieve corresponding improvements in energy by improving the overall compute efficiency. However, Parabix is restricted to the resources available within a single core. In this section we have parallelized the Parabix XML parser by hand to study the effects of thread level parallelism in conjunction with Parabix's data parallelism.

A typical approach to handling data parallelism with multiple threads, requires partitioning data uniformly across the threads. However XML parsing inherently moves through data in sequence, creates data dependencies between the different phases and makes it hard to partition the file. A few attempts have been made to address this problem using a pre-parsing phase to help determine the tree structure and to partition the XML document [15]. There have been other approaches to speculatively partition the data [17] but this requires introducing significant complexity in the overall logic of the program.

We adopt a contrasting approach to parallelizing the Parabix XML parser. As described in Section 4 Parabix consists of multiple passes that on every chunk of input data and each of these stages interact in sequence with no data movement from later to earlier passes. This fits well into the mold of pipeline parallelism and we partition the overall parser into several stages and assign a core to each to stage. One of the key challenges is to evaluate which passes need to be grouped into a stage. We analyzed the latency of each of the passes in the single-threaded version of Parabix XML parser (Column 1 in Table 3) and assigned the passes them to stages such that the stage latency of the overall pipeline is balanced resulting in maximal throughput.

The interface between stages is implemented using a ring buffer, where each entry consists of all ten data structures for one segment as listed in Table 3. Each pipeline stage  $S$  maintains the index of the buffer entry ( $I_S$ ) that is being processed. Before processing the next buffer frame the stage check if the previous stage is done by spinning on  $I_{S-1}$  (Stage  $S-1$ 's buffer entry). In commodity multicore chips typically all



			Data Structures									
			data_buffer	basis_bits	u8	lex	scope	ctCDPI	ref	tag	xml_names	err_streams
	latency(C/B)	size (B)	128	128	496	448	80	176	112	176	16	112
Stage1	1.97	read_data transposition classification	write read	write read		write						
Stage2	1.22	validate_u8 gen_scope parse_CtCDPI parse_ref		read	write	read read read	write read read	write read read	write			write
Stage3	2.03	parse_tag validate_name gen_check			read read	read read read	read read read	read read read	read	write read read	write read	write write
Stage4	1.32	postprocessing	read			read		read	read			read

Table 3: Relationship between Each Pass and Data Structures

threads share the last level cache and if we let faster pipeline stages run ahead the data they process will increase contention to the shared cache. To prevent this we optimize how far the faster pipeline stages can run ahead by controlling the overall size of the ring buffer. The faster stage if it runs ahead will effectively cause the ring buffer to fill up and cause the stage implicitly stall.

Figure 18 demonstrates the performance improvement achieved by pipelined Parabix in comparison with the single-threaded version. The multithreaded is  $\simeq 2\times$  faster compared to the single threaded version and achieves  $\simeq 2.7$  cycles per input byte by exploiting SIMD units of all SandyBridge’s cores. This performance approaches the performance of custom hardware solutions. Parabix demonstrates the potential to enable an entire new class of applications, text processing, to exploit multicores.

Figure 18(b) shows the average power consumed by the multithreaded Parabix. Overall, as expected the power consumption of the multithreaded Parabix increases in proportion to the number of active cores. Note that the increase is not linear, since shared units such as last-level-caches consume active power even if a single core is active. Perhaps more interestingly we achieve corresponding reduction in execution time which leads to same energy consumption as the single-thread version (in some cases marginally less energy as shown for soap.xml).

## 10 Related Work

There has been work in the past which has sought to address the overheads of text processing in specific applications (e.g., XML parsers) and have adopted specialized hardware and software solutions for each

1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455

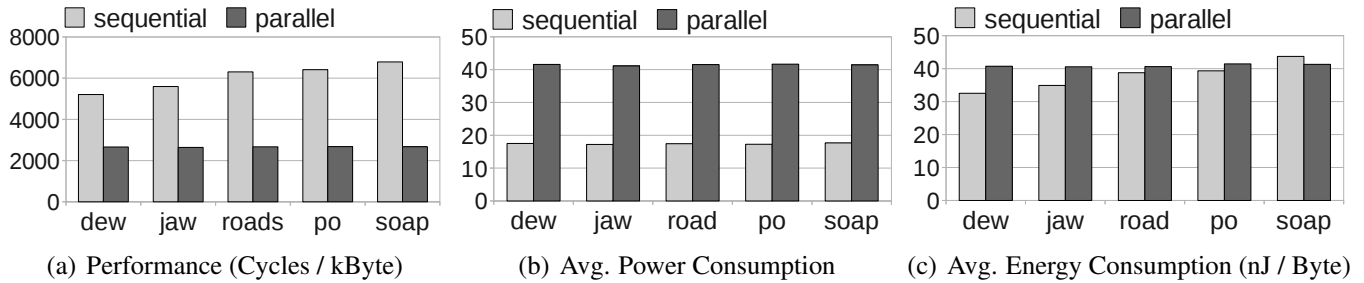


Figure 18: Multithreaded Parabix

application. Nicola and John specifically identified the traditional method of XML parsing as a threat to database performance and outlined a number of potential directions for improving performance [16]. The commercial importance of XML parsing has spurred the development of numerous multi-threaded and hardware-based approaches: Multithreaded XML techniques include preparing the XML file to locate key partitioning points [20] and speculative p-DFAs [20]. Hardware methods include custom XML chips [14] and FPGA-based implementations [10]. Recently Cameron et al. [6, 7] accelerated XML parsing using SSE instructions. Finally, other have explored the design of custom hardware for bit parallel operations in network processors [18].

In this paper, we have introduce parallel bit streams as a general abstraction to parallelize and improve the performance general text processing. We have developed a compiler tool chain and the runtime to enable bit streams to exploit SIMD extensions found on commodity processors. We are also the first to perform a detailed analysis of SIMD instruction extensions across three generations of intel processors including the new 256 bit AVX extensions. Finally, we have shown the benefits of using multithreading in conjunction with data parallel phases of the application.

## 11 Conclusion

In this paper we presented Parabix a software runtime framework for exploiting SIMD data units found on commodity processors for text processing. The Parabix framework allows to focus on exposing the parallelism in their application assuming an infinite resource abstract SIMD machine without worrying about or having to change code to handle processor specifics (e.g., 128 bit SIMD SSE vs 256 bit SIMD on AVX). We applied Parabix technology to a widely deployed application; XML parsing and demonstrate the efficiency gains that can be obtained on commodity processors. Compared to the conventional XML

1456 parsers, Expat and Xerces, we achieve  $2\times$ — $7\times$  improvement in performance and average  $4\times$  improve-  
1457  
1458 ment in energy. We achieve high compute efficiency with an overall  $9\times$ — $15\times$  reduction in branches,  
1459  
1460  $7\times$ — $15\times$  reduction in branch mispredictions, processing upto 128 characters with a single operation. We  
1461  
1462 used the Parabix framework and XML parsers to study the features of the new 256 bit AVX extension  
1463  
1464 in Intel processors. We find that while the move to 3-operand instructions deliver significant benefit the  
1465  
1466 wider operations in some cases have higher overheads compared to the existing 128 bit SSE operations.  
1467  
1468 We also compare Intel's SIMD extensions against the ARM Neon. Note that Parabix allowed us to perform  
1469  
1470 these studies without having to change the application source. Finally, we parallelized the Parabix XML  
1471  
1472 parser to take advantage of the SIMD units in every core on the chip. We demonstrate that the benefits of  
1473  
1474 thread-level-parallelism are complementary to the fine-grain parallelism we exploit; parallelized Parabix  
1475  
1476 achieves a further  $2\times$  improvement in performance.  
1477

## 1478 References

- 1479 [1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf,  
1480 S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report  
1481 UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- 1482 [2] F. Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Depart-  
1483 ment of Computer Sciece, June 2001.
- 1484 [3] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and responsive power models  
1485 for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on*  
1486 *Supercomputing*, ICS '10, pages 147–158, New York, NY, USA, 2010. ACM.
- 1487 [4] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In  
1488 *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, 2010.
- 1489 [5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fifth  
1490 edition). W3C Recommendation, 2008.
- 1491 [6] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich. Parallel scanning with bitstream  
1492 addition: An xml case study. In *Euro-Par 2011, LNCS 6853, Part II*, Lecture Notes in Computer Science, pages 2–13,  
1493 Berlin, Heidelberg, 2011. Springer-Verlag.
- 1494 [7] R. D. Cameron, K. S. Herdy, and D. Lin. High performance XML parsing using parallel bit stream technology. In  
1495 *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages  
1496 222–235, New York, NY, USA, 2008. ACM.
- 1497 [8] J. Clark. The Expat XML Parser. <http://expat.sourceforge.net/>.
- 1498 [9] F. Corporation. Fluke Clamp Meters. <http://www.fluke.com/>.
- 1499 [10] Z. Dai, N. Ni, and J. Zhu. A 1 cycle-per-byte XML parsing accelerator. In *FPGA '10: Proceedings of the 18th Annual*  
1500 *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 199–208, New York, NY, USA, 2010.  
1501 ACM.
- 1502 [11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling.  
1503 In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, 2011.
- 1504 [12] A. S. Foundation. Xerces C++ Parser. <http://xerces.apache.org/xerces-cl/>.
- 1505 [13] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz.  
1506 Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international sympo-*  
1507 *sium on Computer architecture*, ISCA '10, 2010.  
1508  
1509  
1510  
1511

1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567

- [14] M. Leventhal and E. Lemoine. The XML chip at 6 years. In *International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*, Aug. 2009.
- [15] W. Lu, Y. Pan, , and K. Chiu. A parallel approach to xml parsing. *The 7th IEEE/ACM International Conference on Grid Computing*, 2006.
- [16] Matthias Nicola and Jasmi John. XML Parsing: A Threat to Database Performance. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, New Orleans, Louisiana, 2003.
- [17] B. Shah, P. R. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for xml dom parsing. In *Proceedings of the 6th International XML Database Symposium on Database and XML Technologies, XSym '09*, pages 75–90, Berlin, Heidelberg, 2009. Springer-Verlag.
- [18] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005.
- [19] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, ASPLOS '10*, 2010.
- [20] Y. Zhang, Y. Pan, and K. Chiu. Speculative p-DFAs for parallel XML parsing. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 388–397, Dec. 2009.