Parabix : Boosting the Efficiency of Text Processing on Commodity Processors

Dan Lin, Nigel Medforth, Ken Herdy, Arrvindh Shriraman, Rob Cameron
School of Computing Science, Simon Fraser University
{lindanl,nmedfort,ksherdy,ashriram,cameron}@cs.sfu.ca

Abstract

Modern applications employ text files widely for providing data storage in a readable format for applications ranging from database systems to mobile phones. Traditional text processing tools are built around a byte-at-a-time sequential processing model that introduces significant branch and cache miss penalties. Recent work has explored an alternative, transposed representation of text, Parabix (Parallel Bit Streams), to accelerate scanning and parsing using SIMD facilities. This paper advocates and develops Parabix as a general framework and toolkit, describing the software toolchain and run-time support that allows applications to exploit modern SIMD instructions for high performance text processing. The goal is to generalize the techniques to ensure that they apply across a wide variety of applications and architectures. The toolchain enables the application developer to write constructs assuming unbounded character streams and Parabix's code translator generates code based on machine specifics (e.g., SIMD register widths).

The general argument in support of Parabix technology is made by a detailed performance and energy study of XML parsing across a range of processor architectures. Parabix exploits intra-core SIMD hardware and demonstrates 2×–7× speedup and 4× improvement in energy efficiency when compared with two widely used conventional software parsers, Expat and Apache-Xerces. SIMD implementations across three generations of x86 processors are studied including the new SandyBridge. The 256-bit AVX technology in Intel SandyBridge is compared with the well established 128-bit SSE technology to analyze the benefits and challenges of 3-operand instruction formats and wider SIMD hardware. Finally, the XML program is partitioned into pipeline stages to demonstrate that thread-level parallelism enables the application to exploit SIMD units scattered across the different cores, achieving improved performance (2× on 4 cores) while maintaining single-threaded energy levels.

1 Introduction

Modern applications ranging from web search to analytics are mainly data centric operating over large swaths of information. Information expansion and diversification of data has resulted in multiple textual storage formats. Of these, XML is one of the most widely used standards, providing a common framework for encoding and communicating data. It is used in applications ranging from Office Open XML in Microsoft Office to NDFD XML of the NOAA National Weather Service, from KML in Google Earth to Castor XML in the Martian Rovers. To enable these diverse applications we need high performance, scalable, and energy efficient processing techniques for textual data in general, and XML, in particular.

Unfortunately, given the limited levels of parallelism that can be found in traditional text processing, it is not clear how this important class of application can benefit from the growth in multicore processors. As a widely cited Berkeley study [2] reports, text processing applications represented by the "thirteenth dwarf" (parsers/finite state machines) are considered to be the hardest application class to parallelize and process efficiently. Conventional software-based text parsers have many inefficiencies including considerable branch misprediction penalties due to complex input-dependent branching structures as well as poor use of caches and memory bandwidth due to byte-at-a-time processing. ASIC chips that process XML textual data have been around since early 2003, but typically lag behind CPUs in technology due to cost constraints [13].

Parallel bit stream (Parabix) technology is a promising new approach for high performance text processing. The key insight is based on the transposition of byte-oriented character data into parallel bit streams (each with one bit per input byte) which permits text processing to exploit SIMD operations on modern processors [6, 8]. Parabix also incorporates a systematic and portable SIMD programming framework based on our model inductive doubling instruction set architecture [9]. Most recently, a new parallel scanning primitive [7] has been incorporated into the technology base to form the basis of the Pablo compiler documented in this paper.

In this paper, we generalize parallel bit streams and develop the Parabix programming framework to help programmers build text processing applications. Programmers specify operations on unbounded character lists using bit streams in a python environment. Our code generation and runtime system translates them into low-level C++ routines. The Parabix routines exploit the SIMD extensions on commodity processors (SSE/AVX on x86, Neon on ARM) to process hundreds of character positions in an input stream simultaneously, and dramatically improving the execution efficiency. We describe the overall Parabix tool chain, a novel execution framework and software build environment that enables text processing applications to effectively exploit commodity multicores.

We study in detail the performance of Parabix technology in application to the problem of XML parsing on multiple architectures. Figure 1 showcases the overall efficiency of our framework and dramatic improvements in both performance and energy efficiency. The Parabix-XML parser exploits the bit stream technology to dramatically reduce branches in parsing routines and realize a more efficient pipeline execution. It also substantially improves register utilization which minimizes energy wasted on cache misses and data transfers.[1]

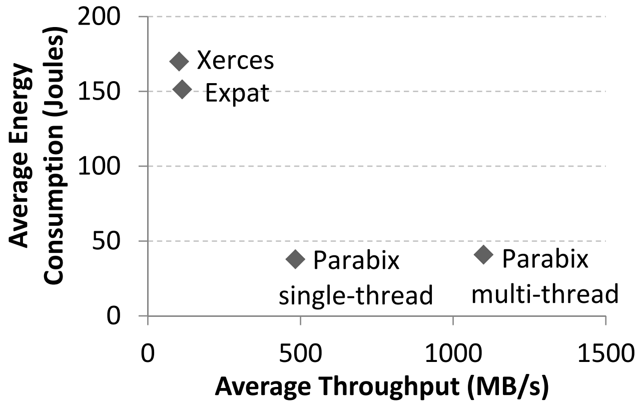


Figure 1: XML Parser Technology Energy vs. Performance

We make the following contributions:

1) We outline the Parabix architecture, code-generation tool chain and runtime environment; and describe how it may be used to produce efficient XML parser implementations on a variety of commodity processors. While studied in the context of XML parsing, the Parabix framework can be widely applied to many problems in text processing and parsing. We have released Parabix as open source and are interested in exploring the applications that can take advantage of our tool chain (*http://parabix.costar.sfu.ca/*).

2) We compare the Parabix XML parser against conventional parsers and assess the improvement in overall performance and energy efficiency on variety of hardware platforms. We use Parabix to study and contrast SSE/AVX extensions across multiple generation of Intel processors and show that there are performance challenges when using newer generation SIMD extensions. We compare the ARM Neon extensions against the x86 SIMD extensions and comment on the latency of SIMD operations.

3) Finally, we multithread the Parabix XML parser to enable the different stages in the parser to exploit SIMD units across all the cores. This further improves performance while maintaining the energy consumption comparable with the sequential version.

The remainder of this paper is organized as follows. Section 2 presents background material on XML parsing and provides insight into the inefficiency of traditional parsers. Section 3 describes the Parabix architecture, tool chain and runtime environment. Section 4 describes the design of an XML parser based on the Parabix framework. Section 5 details our evaluation framework. Section 6 presents a detailed performance analysis of Parabix on a Core-i3 system using hardware performance counters. 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 multithreading to exploit the SIMD units scattered across multiple cores.

[1]The actual energy consumption of the XML ASIC chips is not published by the companies.

2 Background

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.

```
<Products>
  <Product ID="0001">
    <ProductName Lang="English">Widget</ProductName>
    <ProductName Lang="French">Bitoniau</ProductName>
    <Company>ABC</Company>
    <Price>$19.95</Price>
  </Product>
</Products>
```

Figure 2: Sample XML Document

2.1 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 a left angle bracket '<', and the content held within the document. A logical *cursor* position denotes the current character under interpretation. As the parser moves its cursor through the document, it alternates between markup scanning, validation, and content processing operations. In other words, 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 the widely deployed the Apache XML project [1], uses a series of nested switch statements and state-dependent flag tests to control the parsing logic of the program. Xerces's complex data dependent control flow requires between 6–13 branches per byte of XML input, depending on the markup in the file (details in Section 6.2). 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 kilobyte (kB) of XML data. 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 byte-at-a-time 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: (1) a unifying architectural view of text processing in terms of parallel bit streams, (2) a tool chain for automating the generation of parallel bit stream code from higher-level specifications, and (3) a runtime environment, which provides a portable SIMD programming abstraction that is 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 into a transform representation consisting of 8 parallel bit streams, known as *basis bit streams* wherein 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 one-to-one correspondence with the i-th bit 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, $\mathtt{b}_{0\ldots7}$. The bits used to construct $\mathtt{b}_7$ are highlighted in this example.

STRING	b	7	<	A
ASCII	0110001**0**	0011011**1**	0011110**0**	0100000**1**

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
0	1	1	0	0	0	1	**0**
0	0	1	1	0	1	1	**1**
0	0	1	1	1	1	0	**0**
0	1	0	0	0	0	0	**1**

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 logical, shift and arithmetic 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" domains for text. The Parabix framework supports efficient text processing using these two equivalent representations analogous to signal processing that benefits from the use of the frequency domain and time domain.

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

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

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

Character-class bit streams enable up to 128 "comparisons" in parallel through a series of boolean-logic operations [2] that merge multiple basis bit streams into a single character-class stream that marks the positions of key characters. For example, a character is an '<' if and only if $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3 \wedge b_4 \wedge b_5) \wedge \neg(b_6 \vee b_7) = 1$. Addition character classes can be determined with similar formulas. For example, a character is a number `[0-9]` if and only if $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3) \wedge \neg(b_4 \wedge (b_5 \vee b_6))$. An important observation here is that a range of characters can sometimes take fewer operations and require fewer basis bit streams to compute than individual characters. Finding optimal solutions for all character-classes is beyond the scope of this paper.

[2] $\wedge$, $\vee$ and $\neg$ denote the boolean AND, OR and NOT operations.

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 arithmetic operations. Lexical bit streams typically mark multiple current parsing positions. Unlike the single-cursor approach of traditional text parsers, the marking of multiple lexical items allows Parabix to process multiple items 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. A `1` bit in an error stream indicates the presence of a potential error that may require further processing to determine cause and severity.

To form lexical bit streams we introduce two 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*, where *c* denotes an initial set of cursor positions, and *m* denotes a set of "marked" lexical item positions. The ScanThru operation determines the cursor positions immediately following any run of marker positions by calculating $(c+m) \wedge \neg m$. For example, in Figure 4 suppose we have the regular expression `<[a-zA-Z]+>` and wish to find all instances of it in the source text. We begin by constructing the character classes C_0, which consists of all letters, C_1, which contains all '>'s, and C_2, which marks all '<'s. In L_0 the position of every '<' is advanced by one to locate the first character of each token. By computing E_0, the parser notes that "<>" does not match the expected pattern. To find the end positions of each token, the parser calculates L_1 by moving the cursors in L_0 through the letter bits in C_0. L_1 is then validated to ensure that each token ends with a '>' and discovers that "`<error]`" also 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	`<a><val> <str>  <>txt><err]`
$C_0 =$ `[a-zA-Z]`	`.1..111...111.....111..111.`
$C_1 =$ `[>]`	`..1....1.....1...1...1.....`
$C_2 =$ `[<]`	`1..1.....1......1.....1....`
$L_0 =$ `Advance`(C_2)	`.1..1.....1......1.....1...`
$E_0 = L_0 \wedge \neg C_0$	`................1.........`
$L_1 =$ `ScanThru`(L_0, C_0)	`..1....1.....1...1........1`
$E_1 = L_1 \wedge \neg C_1$	`..........................1`

Figure 4: Lexical Parsing in Parabix

Using this parallel bit stream approach, the vast majority of conditional branches used to identify key positions and/or syntax errors at each parsing position are mostly eliminated, which, as Section 6.2 shows, minimizes branch misprediction penalties. Accurate parsing and parallel lexical analysis is done through processor-friendly equations that require neither speculation nor multithreading.

3.2 Parabix Compilers

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

The character class compiler is used to automatically produce bit stream logic for all the individual characters (e.g., delimiters) and character classes (e.g., digits, letters) used in a particular application. Input is specified using a character class syntax adapted from the standard regular expression notations. Output is a minimized set of three-address bitwise operations that compute each of the character classes from the basis bit streams. Figure 5 shows the input and output produced by the character class compiler for the example of `[0-9]` discussed in the previous section. The output operations may be viewed as operations on a single block of input at a time, or may be viewed as operations on unbounded bit streams as supported by the Pablo compiler.

```
 INPUT: digit = [0-9]

OUTPUT: temp1 = (basis_bits.bit_0 | basis_bits.bit_1)
        temp2 = (basis_bits.bit_2 & basis_bits.bit_3)
        temp3 = (temp2 &~ temp1)
        temp4 = (basis_bits.bit_5 | basis_bits.bit_6)
        temp5 = (basis_bits.bit_4 & temp4)
        digit = (temp3 &~ temp5)
```

Figure 5: Character Class Compiler Input/Output

```
INPUT:
def parse_tags(classes, errors):
  classes.C0 = Alpha
  classes.C1 = Rangle
  classes.C2 = Langle
  L0 = bitutil.Advance(C2)
  errors.E0 = L0 &~ C0
  L1 = bitutil.ScanThru(L0, C0)
  errors.E1 = L1 &~ C1

OUTPUT:
struct Parse_tags {
  Parse_tags() { CarryInit(carryQ, 2); }
  void do_block(Classes & classes, Errors & errors){
    BitBlock L0, L1;
    classes.C0 = Alpha;
    classes.C1 = Rangle;
    classes.C2 = Langle;
    L0 = BitBlock_advance_ci_co(C2, carryQ, 0);
    errors.E0 = simd_andc(L0, C0);
    L1 = BitBlock_scanthru_ci_co(L0, C0, carryQ, 1);
    errors.E1 = simd_andc(L1, C1);
    CarryQ_Adjust(carryQ, 2);
  }
  CarryDeclare(carryQ, 2);
};
```

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

The Pablo compiler abstracts away the details of programming parallel bit stream code in terms of finite SIMD register widths and application buffer sizes. Input to Pablo is a language for expressing bit stream operations on unbounded bit streams. The operations include bitwise logic, the `Advance` and `ScanThru` 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` and `CarryInit` declare and initialize the Carry Queue structure depending on the specific architecture and Carry Queue representation. The unbounded bit stream `Advance` and `ScanThru` operations are translated into block-wise equivalents with explicit carry-in and carry-out processing. At the end of each block, the `CarryQ_Adjust` operation implements any necessary adjustment of the Carry Queue to prepare for the next iteration. The Pablo language and compiler also support conditional and iterative bit stream logic on unbounded streams (if and while constructs) which involves additional carry-test insertion in control branches. A complete explanation of the Pablo translation is beyond the scope of this paper.

3.3 Parabix Runtime Libraries

The Parabix architecture also includes runtime libraries that support a machine-independent view of basic SIMD operations, as well as a set of core function libraries. For portability, we program all SIMD operations against an abstract SIMD machine representation, parameterized on SIMD field and register width. The abstract machine supports all power-of-2 field widths up to the full SIMD register width on a target machine. Let $w = 2k$ be the field width in bits. Let f be a basic binary operation defined on w-bit quantities producing an w-bit result. Let W be the SIMD vector size in bits where $W = 2K$. Then the C++ template notation `v=simd<w>::f(a,b)` denotes the general pattern for a vertical SIMD operation yielding an output SIMD vector v, given two input SIMD vectors a 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, `simd<8>::add(a,b)` represents the simultaneous addition of sixteen 8-bit fields.

We have ported Parabix to a wide variety of processor architectures demonstrating its applicability to commodity SIMD hardware. We currently take advantage of the 128-bit Altivec operations on the Power PC, 64-bit MMX and 128-bit SSE operations on previous generation Intel platforms, the latest 256-bit AVX extensions on the SandyBridge processor, and finally the 128-bit Neon operations on ARM.

4 The Parabix XML Parser

This section describes the implementation of the Parabix XML parser for well-formedness checking. Figure 7 shows its overall structure. The input file is processed using 11 functions organized into 7 modules. In the first module, `Read_Data`, the input file is loaded into the data_buffer. The data is then transposed to eight parallel basis bit streams (basis_bits) in the `Transposition` module. The basis_bits are used in the `U8_Validation` module to validate UTF-8 characters, and by the `Classification` and `Gen_Scope` module to generate all the XML lexical item streams (lex) and 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. Afterward, 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 runtime library, as well as the hand-written postprocessing code that completes the well-formed checking.

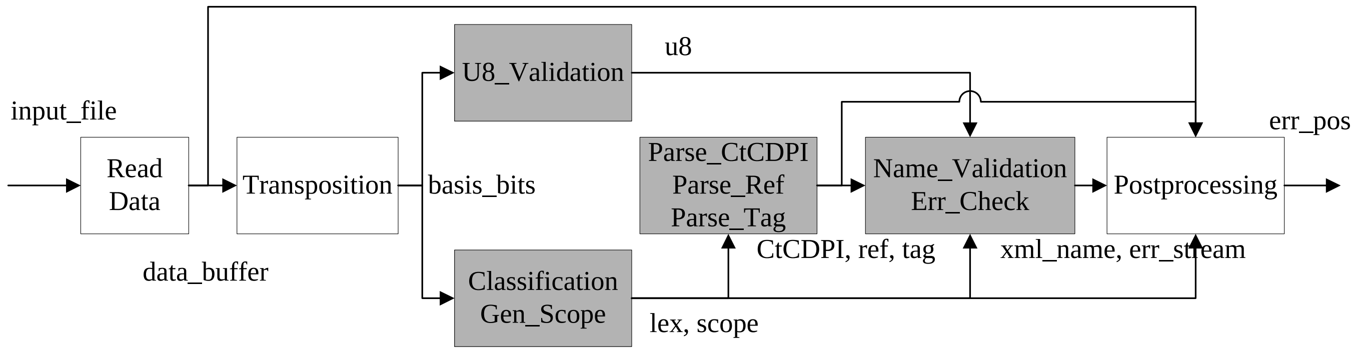


Figure 7: Parabix XML Parser Structure

5 Evaluation Framework

XML Parsers: We evaluate the Parabix XML parser described above against two widely available open-source parsers: Xerces-C [1] and Expat [10]. 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) is a validating XML parser written in C++ and is available as part of the the Apache project. Expat version 2.0.1 is a stream-oriented non-validating XML parser library written in C. To ensure a fair comparison, we restricted our analysis of Xerces-C to its WFXML scanner to eliminate the cost of non-well-formedness validation and used the SAX interface to avoid the memory cost of DOM tree construction.

XML Workloads: XML is used for a variety of purposes ranging from databases to configuration files in mobile phones. A key predictor of the overall parsing performance of an XML file is *Markup density* (i.e., the ratio of markup vs. the total XML document size.) This metric has substantial influence on the performance of traditional recursive descent XML parsers. We use a mixture of document-oriented and data-oriented XML files to analyze performance over a 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.

File Name	dew.xml	jaw.xml	roads.gml	po.xml	soap.xml
File Type	doc	doc	data	data	data
File Size (kB)	66240	7343	11584	76450	2717
Markup Density	0.07	0.13	0.57	0.76	0.87

Table 1: XML Document Characteristics

Platform Hardware: SSE SIMD 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, register resources, and the addition of domain specific instructions. Here we investigate SIMD extensions across three different generations of intel processors: Core2Duo (2.13Ghz,32KB L1, 2MB Shared L2), Core i3 (2.9Ghz, 32KB L1,256KB L2, 4MB Shared LLC), and Sandybridge (2.8Ghz, 32KB L1, 256KB L2, 6MB LLC). We compare the energy and performance profile of the Parabix under the platforms. We also analyze the implementation specifics of SIMD extensions under various microarchitectures and the newer AVX extensions. We investigate the execution profiles of each XML parser using the performance counters found in the processor. We choose several key hardware events that provide insight into the profile of each application and indicate if the processor is doing useful work [3, 4]. The set of events 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 framework.

Energy Measurement: We measure the energy consumption of the processor directly using a current clamp. We apply the Fluke i410 current clamp 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 the 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 $12V * \sum_{i=1}^{N_{samples}} Sample_i$.

6 Efficiency of Parabix-XML

In this section we analyze the energy and performance characteristics of the Parabix XML parser against the software XML parsers, Xerces and Expat, each applied to the problem of XML well-formedness checking. For our baseline evaluation, we compare all the XML parsers on the Core-i3.

6.1 Cache behavior

Table 2 shows 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 cost.[3]

	Parabix	Expat	Xerces
L1	4.1	31.7	104.2
L2	0.1	12.0	1.7
L3	0.03	3.9	0.3

Table 2: Cache Misses per kB of input data

This overhead has little impact on the overall performance of these parsers as they experience additional overheads related to branch mispredictions. When compared with Xerces and Expat, the data organization of Parabix-XML significantly reduces the overall cache miss rate; specifically, there were 7× and 15× fewer L1 and L2 cache misses compared to the next best parser tested. The improved cache utilization helps keep the SIMD units busy by minimizing memory-related stalls and lowers the overall energy consumption by reducing the need to access the higher levels of the cache hierarchy. Using microbenchmarks, we estimated that the L1, L2, and L3 cache misses consume ∼8.3nJ, ∼19nJ, and ∼40nJ respectively. On average, with 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

The performance of traditional parsers is limited by their branch behavior. Xerces experiences up to 13 branches per input XML character on the high markup files whereas Expat experiences up to 8. In Parabix-XML, the use of SIMD operations eliminates a significant proportion of the overall branches. Most conditional branches can be replaced with bitwise operations, which can process up to 128 characters worth of branches with one operation or with a series of logical predicate operations, which are cheaper to compute since they require only SIMD operations.

The high branch misprediction rate of conventional parsers is a significant overhead, with the cost of a single branch misprediction on the order of 10s of CPU cycles spent to restart the processor pipeline. Parabix-XML is nearly branch free and exhibits minimal dependence on the source markup density. Specifically, our study demonstrates that Parabix experiences between 19.5 and 30.7 branch mispredictions per kB of XML data. Conversely, the cost of branch mispredictions for the Expat parser can be over 7 cycles per XML byte (see Figure 8) — which exceeds the average latency of a byte processed by Parabix-XML.

The branch misprediction rate of traditional XML parsers is difficult to reduce due to a number of factors: (1) the variable length nature of the syntactic elements contained within XML documents; (2) input data dependent characteristic, and (3) the extensive set of syntax constraints imposed by the XML specifications.

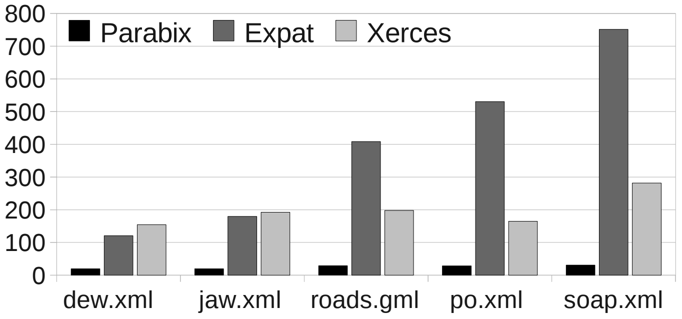


Figure 8: Branch Mispredictions on the Core-i3 per kB input

6.3 SIMD Instructions vs. Total Instructions

In Parabix-XML, the ratio of retired SIMD instructions to total instructions provides insight into the relative degree to which Parabix-XML achieves parallelism over the byte-at-a-time approach. Using the Intel Pin tool, we gathered the dynamic instruction mix for each XML workload and classified the instructions as either SIMD or non-SIMD. Figure 3 shows the percentage of SIMD instructions in the Parabix-XML parser. The ratio of executed SIMD instructions over total instructions indicates the amount of available parallelism. The resulting instruction mix consists 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.

File Name	dew.xml	jaw.xml	roads.gml	po.xml	soap.xml
SIMD	81.68%	80.59%	70.7%	66.02%	59.9%
Non-SIMD	18.32%	19.41%	29.3%	33.98%	40.1%

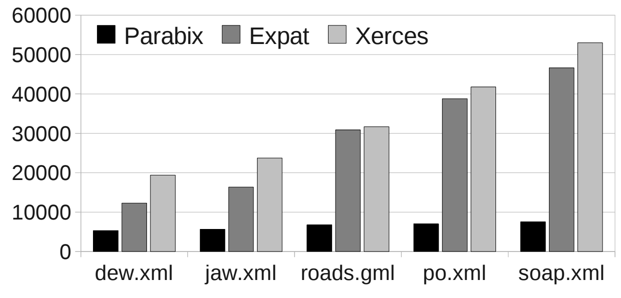
Table 3: SIMD Instruction Percentage

6.4 Performance and Energy Characteristics

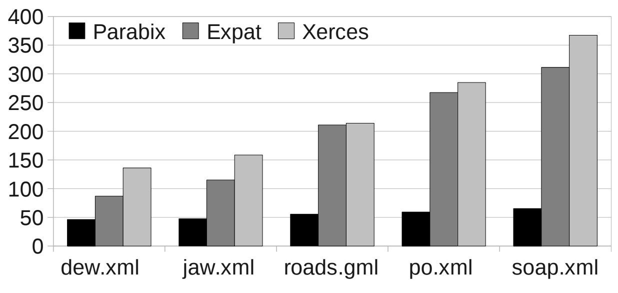
Figure 9(a) shows overall parser performance in terms of CPU cycles per kB. Parabix-XML is 2.5 to 4× faster on document-oriented input and 4.5 to 7× faster on data-oriented input. Traditional parsers can be dramatically slowed by dense markup but Parabix-XML is relatively unaffected. Unlike Parabix-XML and Expat, Xerces transcodes input to UTF-16 prior to processing; this requires several cycles per byte. However, transcoding using parallel bit streams is significantly faster and requires less than a single cycle per byte.

Parabix consumes substantially less energy (see Figure 9(b)) than the other parsers. Parabix consumes 50 to 75 nJ per byte while Expat and Xerces consume 80nJ to 320nJ and 140nJ to 370nJ per byte respectively. Parabix-XML experiences minimal increase in power consumption (∼ 5%) as

[3]The approximate miss penalty on the Core-i3 for L1, L2 and L3 caches is 4, 11, and 36 cycles respectively.



(a) Performance (CPU Cycles per kB)



(b) Energy Consumption (μJ per kB)

Figure 9: Performance and Energy profile of Parabix on Core i3

compared to the conventional parsers. 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 pipeline processor. Parabix amortizes the fetch and data access overheads over multiple data parallel operations. Although Parabix requires slightly more power (per instruction), the processing time of Parabix is significantly lower resulting in an overall improvement in energy. The Parabix parser makes efficient use of the processor pipeline which minimizes overall energy wastage.

7 Parabix on different platforms

7.1 Performance

In this section, we study the performance of the XML parsers across three generations of Intel architectures. Figure 10 shows the average execution time of Parabix-XML over all workloads. We analyze the execution time in terms of SIMD operations that operate on "bit streams" in *bit-space* and scalar operations used to perform "post processing" operations on the source input.

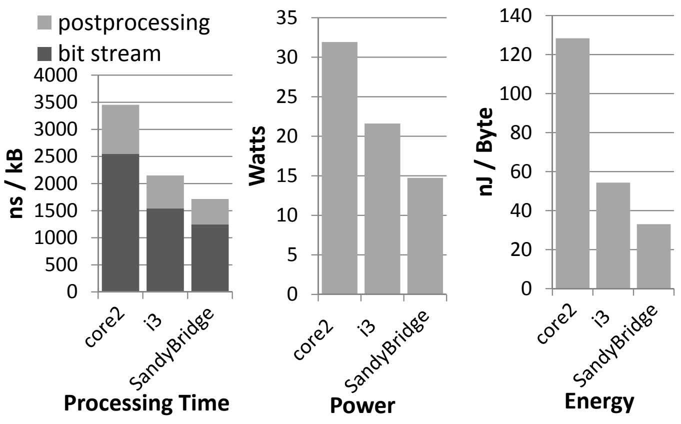


Figure 10: Parabix on various hardware platforms

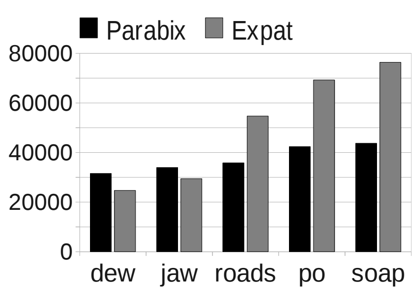
Our results demonstrate that Parabix-XML's optimizations complement newer hardware improvements. For bit stream processing, Core-i3 has a 40% performance increase over Core2; similarly, SandyBridge has a 20% improvement compared to Core-i3. These gains appear independent of the markup. Postprocessing operations demonstrate data dependent variance. Performance on the Core-i3 increases by 27%–40% compared to Core2 whereas SandyBridge increases by 16%–29% compared to Core-i3. Core-i3 improves performance only by 29% over Core2 while SandyBridge improves performance by less than 6% over Core-i3. Note that the gains of Core-i3 over Core2 includes an improvement both in clock frequency and microarchitecture while SandyBridge's gains are mainly attributed to the architecture. Figure 10 also shows the average power consumption of Parabix-XML over each workload and as executed on each of the processors: Core2, Core-i3 and SandyBridge. Each generation of processor appears to bring a 25–30% improvement in power consumption over the previous generation. Parabix-XML on SandyBridge consumes 72%–75% less energy than it did on Core2.

7.2 Parabix on Mobile Processors

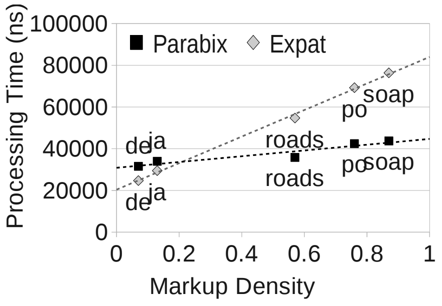
Our experience with Intel processors led us to question whether mobile processors with SIMD support, such as the ARM Cortex-A8, could benefit from Parabix technology. ARM Neon provides a 128-bit SIMD instruction set similar in functionality to the Intel SSE3 instruction set. In this section, we present 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++ applications.

The platform we use is the Samsung Galaxy Android Tablet that houses a Samsung S5PC110 ARM Cortex-A8 1Ghz single-core, dual-issue, superscalar microprocessor. This device includes a 32kB L1 data cache and a 512kB L2 shared cache. Migration of Parabix-XML to the Android platform only required developing a Parabix runtime library for ARM Neon. The majority of the runtime functionality was ported directly. However, a small subset of key SIMD instructions (e.g., bit packing) did not exist on Neon. In such cases, the logical equivalents of those instructions were emulated using the available ISA. The resulting application was cross-compiled for Android using the Android NDK.

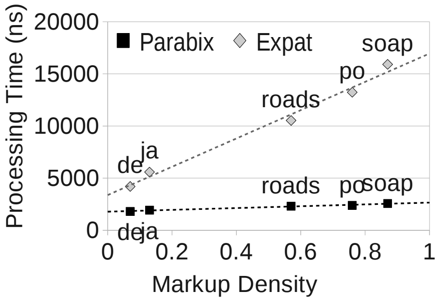
A comparison of Figure 11(a) and Figure 9(a) demonstrates that the performance of both Parabix and Expat de-



(a) ARM Neon Performance (cycles per kB)



(b) ARM Neon



(c) Core i3

Figure 11: Comparison of Parabix-XML on ARM vs. Intel.

grades substantially on Cortex-A8 (5–17×). This result was expected given the comparably performance limited Cortex-A8. Surprisingly, on Cortex-A8, Expat outperforms Parabix on each of the lower markup density workloads, dew.xml and jaw.xml. On the remaining higher-density workloads, Parabix performs only moderately better than Expat. Investigating causes for this performance degradation for Parabix led us to investigate the latency of Neon SIMD operations.

Figure 11(b) investigates the performance of Expat and Parabix for the various input workloads on the Cortex-A8; Figure 11(c) plots the performance for Core-i3. The results demonstrate that the execution time of each parser varies in a linear fashion with respect to the markup density of the file. On the both Cortex-A8 and Core-i3 both parsers demonstrate the same trend: files with a lower markup density exhibit higher levels of parallelism; consequently, the overhead of SIMD instructions has a greater impact on the overall execution time for those files. The contrast between Figure 11(b) and 11(c) provides insight into the problem: Parabix-XML's performance is hindered by SIMD instruction latency. This is possibly because the Neon SIMD extensions are implemented as a coprocessor on the Cortex-A8, which imposes a higher overhead for applications that frequently inter-operate between scalar and SIMD registers. Future performance enhancements to the Neon ISA on ARM could substantially improve the efficiency of Parabix.

8 Parabix on AVX

In this section, we discuss the scalability and performance advantages of our 256-bit AVX (Advanced Vector Extensions) Parabix-XML port. The Parabix runtime libraries originally targeted the 128-bit SSE2 SIMD technology, available on all modern 64-bit Intel and AMD processors. It was recently been ported to AVX, which is commercially available on the latest the SandyBridge microarchitecture Intel processors. Although the runtime had to be ported to the new ISA, no modifications were made to the application.

8.1 3-Operand Form

In addition to widening the 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. As such, 2-operand instructions that require the value of both *a* and *b*, must either copy an additional register value beforehand, or reconstitute a register 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. By avoiding the need to copy or reconstitute operand values, a considerable reduction in instructions required for unloading from and loading into registers is achieved. AVX technology makes available the 3-operand form for both the new 256-bit AVX as well as the 128-bit SSE operations.

8.2 256-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 SandyBridge AVX implementation, Intel has focused primarily on floating point 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.

8.3 Performance Results

We implemented two versions of Parabix-XML using AVX technology. The first was simply the recompilation of the existing Parabix-XML source code 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 Parabix runtime library to leverage the 256-bit AVX instructions wherever possible and to simulate the remaining operations using pairs of 128-bit operations. Figure 12 shows the reduction in instruction count achieved in each version. For each workload, the base instruction count of the Parabix binary compiled in 2-operand SSE-only mode is indicated by "sse"; the version that only takes advantage of the AVX 3-operand mode is labeled "128-bit avx", and the version uses the 256-bit operations wherever possible is labeled "256-bit avx". The instruction counts are divided into three classes: "non-SIMD" operations are the general purpose instructions.

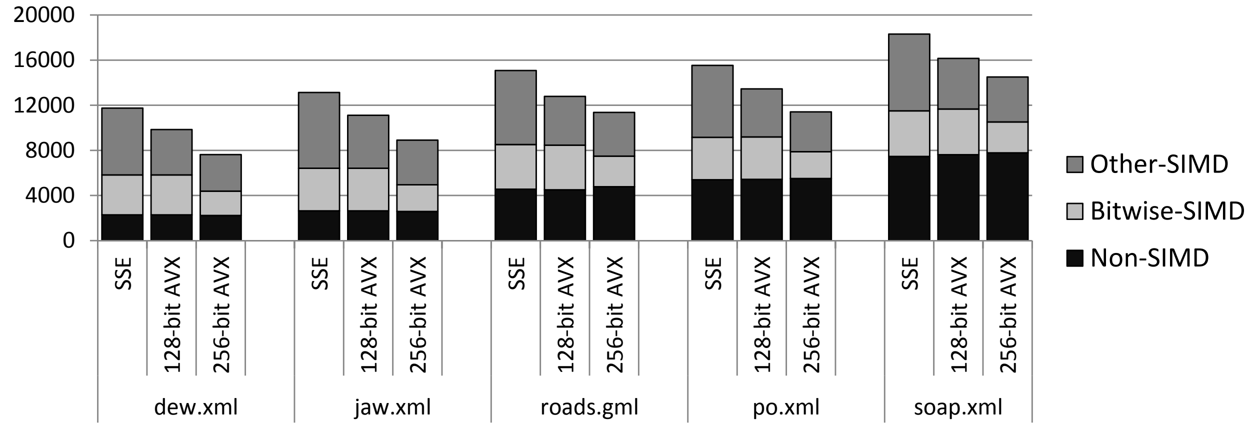


Figure 12: Parabix Instruction Counts (y-axis: Instructions per kB)

The "bitwise SIMD" class comprises the bitwise logic operations, that are available in both 128-bit form and 256-bit form — excluding bitwise shifts which are only available in 128-bit form. The "other SIMD" class comprises all other SIMD operations, primarily comprising the integer SIMD operations that are available only at 128-bit widths even under AVX.

The number of non-SIMD instructions remains relatively constant with each implementation. The number of bitwise SIMD operations remains the same for both SSE and 128-bit AVX while dropping dramatically when operating 256-bits at a time. The reduction was measured at 32%–39% depending on markup density of the workload. The "other SIMD" class shows a substantial 30%–35% reduction with AVX 128-bit technology compared to SSE. This reduction is due to elimination of register unloading and reloading when SIMD operations are compiled using 3-operand AVX form versus 2-operand SSE form. A further 10%–20% reduction is also observed when Parabix-XML utilized the AVX runtime library.

The reductions in instruction counts are significant with the AVX extensions demonstrating the ability of Parabix to exploit wider SIMD extensions. Figure 13 shows the benefits of the reduced SIMD instruction count are achieved only in the AVX 128-bit version; The 3-operand form seems to fully translate to performance benefits. Based on the reduction of overall Bitwise-SIMD instructions we expected a 11% improvement in performance. Surprisingly, the performance of Parabix in the 256-bit AVX implementation does not improve significantly and actually degrades for files with higher markup density ($\sim 11\%$). dew.xml, on which bitwise-SIMD instructions were reduced by 39%, saw a performance improvement of 8%. We believe that this is primarily due to the intricacies of the first generation AVX implementation in SandyBridge, with significant latency in many of the 256-bit instructions in comparison to their 128-bit counterparts. The 256-bit instructions also have different scheduling constraints that seem to reduce overall throughput. If these latency issues can be addressed in future AVX implementations, further performance and energy benefits could be realized by Parabix.

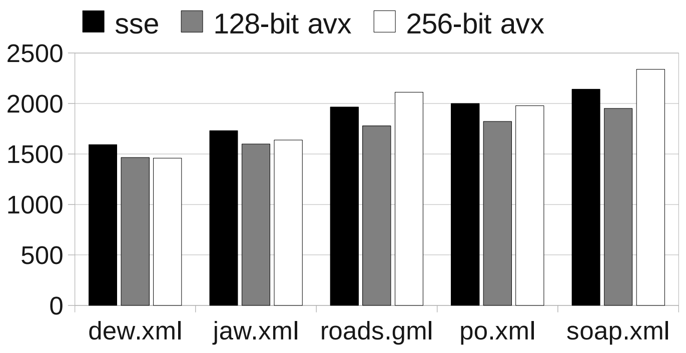


Figure 13: Parabix Performance (y-axis: ns per kB)

9 Multithreaded Parabix

Even if an application is infinitely parallelizable and thread synchronization costs are non-existent, all applications are constrained by the power and energy overheads incurred when utilizing multiple cores; as more cores are put to work, a proportional increase in power occurs. 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-XML can improve performance and reduce energy consumption by improving the overall computation efficiency. In this section, we discuss our parallelized version of Parabix-XML to study the effects of thread-level parallelism in conjunction with Parabix-XML's data parallelism.

The typical approach to handling data parallelism with multiple threads involves partitioning data uniformly across the threads. However, XML parsing is inherently sequential, which makes it difficult to partition the data. Several attempts have been made to address this problem. For example, using a preparsing phase to help determine the tree structure and to partition the XML document accordingly [15]. Another approach involved speculatively partitioning the data [18] but this introduced a significant level of complexity into the overall logic of the program.

In contrast to those methods, we adopted a parallelism strategy that requires neither speculation nor pre-parsing. As

	functions	latency(C/B)
Stage1	read_data, transposition, classification	1.97
Stage2	validate_u8, gen_scope, parse_CtCDPI, parse_ref	1.22
Stage3	parse_tag validate_name gen_check	2.03
Stage4	postprocessing	1.32

Table 4: Stage Division

described in Section 4, Parabix-XML consists of multiple passes that operate on every chunk of input data, and interact with each other in sequence with no data movement from later to earlier passes. This fits well into the mold of pipeline parallelism. We partitioned Parabix-XML into four stages and assigned a core to each to stage. One of the key challenges was to determine which passes should be grouped together. We analyzed the latency and data dependencies of each of the passes in the single-threaded version of Parabix (Column 3 in Table 4), and assigned the passes to stages to maximize throughput.

The interface between stages is implemented using a ring buffer, where each entry consists of all ten data structures for one segment. 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 threads share the last level cache. If we let the faster pipeline stages run ahead, the data they process will increase contention to the shared cache. To prevent this we limit how far the faster pipeline stages can run ahead by controlling the overall size of the ring buffer. Whenever a faster stage runs ahead, it will effectively cause the ring buffer to fill up and force that stage to stall. Experiments show that six entries of the circular buffer gives the best performance.

Figure 14 demonstrates the performance improvement achieved by pipelined Parabix-XML in comparison with the single-threaded version. The 4-threaded version is $\simeq 2\times$ faster compared to the single threaded version and achieves $\simeq 2.7$ cycles per input byte by exploiting the SIMD units of all SandyBridge's cores. This performance approaches the 1 cycle per byte performance of custom hardware solutions [11]. Parabix demonstrates the potential to enable an entire new class of applications, text processing, to exploit multicores.

Figure 14 also shows the average power consumed by the multithreaded Parabix. Overall, as expected the power consumption 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 only one core is active. Perhaps more interestingly there is a reduction in execution time, which leads to the energy consumption being similar to the the single-thread execution (in some cases marginally less energy e.g., 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

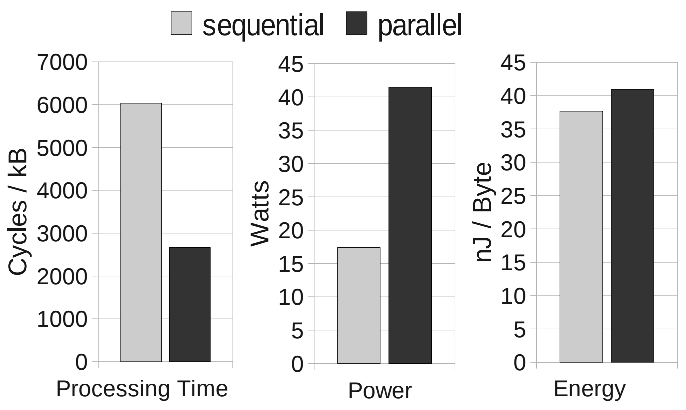


Figure 14: Average Statistic of Multithreaded Parabix

(e.g., XML parsers) and have adopted specialized hardware and software solutions for each application. XML parsing as a threat to database performance [16] outlines a number of potential directions for improving performance. The commercial importance of XML parsing has spurred the development of numerous multi-threaded and hardware-based approaches: Multithreaded XML techniques include preparsing the XML file to locate key partitioning points [14, 18] and speculative p-DFAs [20]. Hardware methods include custom XML chips [13] and FPGA-based implementations [11]. Others have explored the design of custom hardware for bit parallel operations for text search in network processors [19]. Intel's SSE4.2 instructions targeted XML parsers, but these have not seen widespread use because of portability concerns and the programming challenges that accompany low level instructions [12].

Parallel bitstream methods were first introduced in application to the problem of UTF-8 to UTF-16 transcoding [6]. Our first XML parser used SSE2 instructions for bitstream transposition and character class formation coupled with processor bit scan operations to accelerate the sequential scans in a recursive descent parser [8]. The technique has also been employed to accelerate string matching operations in protein identification [17]. The work on an inductive doubling instruction set [9] established the basis for our portable SIMD run-time environments, while the introduction of parallel scanning primitives [7] has provided the basis for our Pablo compiler technology on unbounded bitstreams.

In this paper, we have developed a generalized Parabix architecture and have described the software tool chain that programmers can use to build scalable text processing applications on commodity multicores. We have explored in the detail the tradeoffs between the SIMD implementations across processor generations (i.e., SSE vs AVX) and multiple platfoms (ARM vs Intel). Finally, we have also explored the benefits of using pipeline-based multicore parallelism as a technique to eliminate imbalances in SIMD bitstream-based parallelization and improve overall efficiency.

11 Conclusion

This paper presents Parabix as a software runtime framework for exploiting SIMD data units found on commodity processors for text processing. The Parabix framework allows programmers 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). Parabix technology was applied to XML parsing to demonstrate the efficiency gains that can be obtained on commodity processors. Compared to the conventional XML parsers, Expat and Xerces, a 2×—7× improvement in performance and average 4× improvement in energy was achieved. Furthermore, computational efficiency was greatly increased, with an overall 9×—15× reduction in branches and 7×—15× reduction in branch mispredictions.

The Parabix framework and XML parsers was also used to study the features of the new 256-bit AVX extension in Intel processors. While the move to 3-operand instructions delivers significant benefits, the advantage of loads and bitwise logic with 256 bits at a time was negated by the need to convert to 128 bit SIMD registers for integer operations. We expect this will be remedied with AVX2. Intel's SIMD extensions were also compared with the ARM Neon. Note that Parabix allowed us to perform these studies without having to change the application source. Finally, the Parabix XML parser was parallelized to take advantage of the SIMD units in every core on the chip, demonstrating that the benefits of thread-level-parallelism are complementary to the fine-grain parallelism we exploit. In this study, our parallelized Parabix achieves a further 2× improvement in performance.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd, Martha Kim, for suggestions and feedback that helped to improve this paper. This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and Simon Fraser University.

References

[1] Apache Software Foundation. Xerces C++. http://xerces.apache.org/xerces-c/.

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

[3] F. Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Sciece, June 2001.

[4] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proc. 24th ACM Int'l Conference on Supercomputing*, ICS '10, pages 147–158, New York, NY, USA, 2010. ACM.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, 2008.

[6] R. D. Cameron. A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 91–98, New York, NY, USA, 2008. ACM.

[7] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par 2011, LNCS 6853, Part II*, Lecture Notes in Computer Science, pages 2–13, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] R. D. Cameron, K. S. Herdy, and D. Lin. High performance XML parsing using parallel bit stream technology. In *CASCON '08: Proc. 2008 conference of the center for advanced studies on collaborative research*, pages 222–235, New York, NY, USA, 2008. ACM.

[9] R. D. Cameron and D. Lin. Architectural support for SWAR text processing with parallel bit streams: the inductive doubling principle. In *ASPLOS '09: Proc. 14th Int'l conference on Architectural support for programming languages and operating systems*, pages 337–348, New York, NY, USA, 2009. ACM.

[10] J. Clark. The Expat XML Parser. http://expat.sourceforge.net/.

[11] Z. Dai, N. Ni, and J. Zhu. A 1 cycle-per-byte XML parsing accelerator. In *FPGA '10: Proc. 18th Annual ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays*, pages 199–208, New York, NY, USA, 2010. ACM.

[12] Z. Lei. XML parsing accelerator with Intel streaming SIMD extensions 4 (Intel SSE4). Intel Software Network, Dec. 2008.

[13] M. Leventhal and E. Lemoine. The XML chip at 6 years. In *Int'l Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*, Aug. 2009.

[14] X. Li, H. Wang, T. Liu, and W. Li. Key elements tracing method for parallel XML parsing in multi-core system. *Int'l Conference on Parallel and Distributed Computing Applications and Technologies*, 0:439–444, 2009.

[15] W. Lu, Y. Pan, , and K. Chiu. A parallel approach to XML parsing. In *The 7th IEEE/ACM Int'l Conference on Grid Computing*, 2006.

[16] Matthias Nicola and Jasmi John. XML Parsing: A Threat to Database Performance. In *Proc. Twelfth Int'l Conference on Information and Knowledge Management*, New Orleans, Louisiana, 2003.

[17] R. J. Peace, H. A. Mahmoud, and J. R. Green. Exact string matching for MS/MS protein identification using the Cell Broadband Engine. *Journal of Medical and Biological Engineering*, 31(2):99–104, 2011.

[18] B. Shah, P. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for XML DOM parsing. In Z. Bellahsne, E. Hunt, M. Rys, and R. Unland, editors, *Database and XML Technologies*, volume 5679 of *Lecture Notes in Computer Science*, pages 75–90. Springer Berlin / Heidelberg, 2009.

[19] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proc. 32nd Annual Int'l Symposium on Computer Architecture*, 2005.

[20] Y. Zhang, Y. Pan, and K. Chiu. Speculative p-DFAs for parallel XML parsing. In *Int'l Conference on High Performance Computing (HiPC)*, pages 388–397, Dec. 2009.