Parabix : Boosting the Efficiency of Text Processing on Commodity Processors

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 Bit Stream) 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 bit stream 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×–7× speedup and 4× 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 SandyBridge. 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× 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 a larger number of cores. Unfortunately, given the limited levels of parallelism that can be found in applications [4], it is not certain how many cores can be productively used in scaling our chips [11]. This is because exploiting parallelism across multiple cores tends to require heavy weight threads that are difficult to manage and synchronize.

The desire to improve the overall efficiency of computing is pushing designers to explore customized hardware [13, 23] that accelerate specific parts of an application while reducing the overheads present in general-purpose processors. They seek to exploit the transistor bounty to provision many different accelerators and keep only the accelerators needed for an application active while switching off others on the chip to save power consumption. While promising, given the fast evolution of languages and software, its hard to define a set of fixed-function hardware for commodity processors. Furthermore, the toolchain to create such customized hardware is itself a hard research challenge. We believe that software, applications, and runtime models themselves can be refactored to significantly improve the overall computing efficiency of commodity processors.

In this paper we tackle the infamous "thirteenth dwarf" (parsers/finite state machines) that is considered to be the hardest application class to parallelize [1]. We present Parabix, a novel execution framework and software runtime environment that can be used to dramatically improve the efficiency of text processing and parsing on commodity processors. Parabix transposes byte-oriented character data into parallel bit streams and then exploits the SIMD extensions on commodity processors (SSE/AVX on x86, Neon on ARM) to process hundreds of character positions in an input stream simultaneously. To transform character-oriented data into bit streams Parabix exploits sophisticated SIMD instructions that enable data elements to be packed into registers. This improves the overall cache behaviour of the application resulting in significantly fewer misses and better utilization. Parabix also dramatically reduces branches in the parsing routines resulting in a more efficient pipeline and substantially improves register utilization which minimizes energy wasted on data transfers.

We apply Parabix technology to the problem of XML parsing. XML is a standard of the web consortium that provides a common framework for encoding and communicating data. XML provides critical data storage for 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.

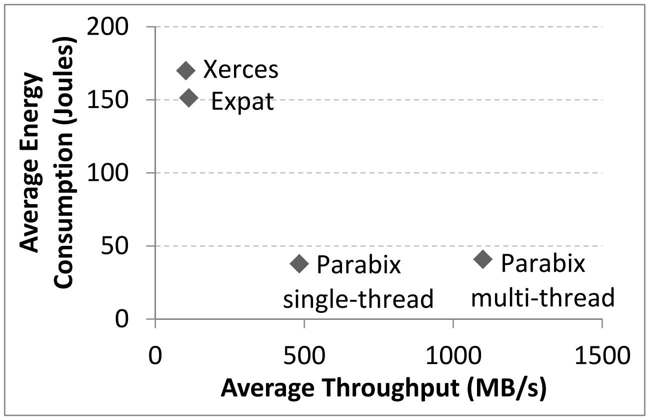


Figure 1: XML Parser Technology Energy vs. Performance

XML parsing efficiency is important for multiple application areas; in server workloads the key focus in on overall transactions per second, while in applications for network switches and cell phones, latency and energy are of paramount importance. Conventional software-based XML 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. XML ASIC chips have been around since early 2003, but typically lag behind CPUs in technology due to cost constraints [16]. They also focus mainly on speeding up the parser computation itself and are limited by the poor memory behaviour. Our focus is how much we can improve performance of the XML parser on commodity processors with Parabix technology.

Figure 1 showcases the overall efficiency of our framework. The Parabix-XML parser improves the performance and energy efficiency several-fold compared to widely-used software parsers, approaching the performance of ASIC XML parsers [10, 16]. [1] Overall we make the following contributions:

1) We outline the Parabix architecture, 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 completely 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

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

overall performance and energy efficiency on variety of hardware platforms. We are the first to compare 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, building on the SIMD parallelism of Parabix technology, we multithread the Parabix XML parser to 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 constant 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 our design of an XML parser based on the Parabix 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 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.

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

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. 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 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: (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 run-time 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 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, $\mathtt{b}_{0\ldots7}$. The bits used to construct $\mathtt{b}_7$ have been 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 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 these two equivalent representations in the same way that efficient signal processing benefits from the use of the frequency domain in some cases and the time domain in others.

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 that we are starting a new 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 allow us to perform up to 128 “comparisons” in parallel with a single operation by using a series of boolean-logic operations [2] to merge multiple basis bit streams into a single character-class stream that marks the positions of key characters with a 1. For example, a character is an ‘<’ if and only if $\neg(b_0 \vee b_1) \wedge (b_2 \wedge b_3 \wedge b_4 \wedge b_5) \wedge \neg(b_6 \vee b_7) = 1$. Classes of characters can be found 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 to all character-classes is non-trivial and goes 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 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 $(c+m) \land \neg m$. 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]`" 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                        <a><valid> <string>  <>ignored><error]
C0 = [a-zA-Z]                      .1..11111...111111.....1111111..11111.
C1 = [>]                           ..1......1........1...1.......1.......
C2 = [<]                           1..1.......1.........1.........1......
L0 = Advance(C2)                   .1..1.......1.........1.........1.....
E0 = L0 ∧ ¬C0                      ......................1...............
L1 = ScanThru(L0, C0)              ..1......1........1...1..............1
E1 = L1 ∧ ¬C1                      .....................................1
```

Figure 4: Lexical Parsing in Parabix

Using this parallel bit stream approach, conditional branch statements used to identify key positions and/or syntax errors at each 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 to compute each of the character classes from the basis bit streams.

For example, 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

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`

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

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-by-block 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. Explaining the full details of the translation is beyond the scope of this paper.

3.3 Parabix Run-Time Libraries

The Parabix architecture also includes run-time libraries that support a machine-independent view of basic SIMD operations, as well as a set of core function libraries. For machine-independence, we program all operations using an abstract SIMD machine. 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. Figure 7 shows its overall structure set up for well-formedness checking. 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 by the `U8_Validation` module to validate UTF-8 characters, and by the

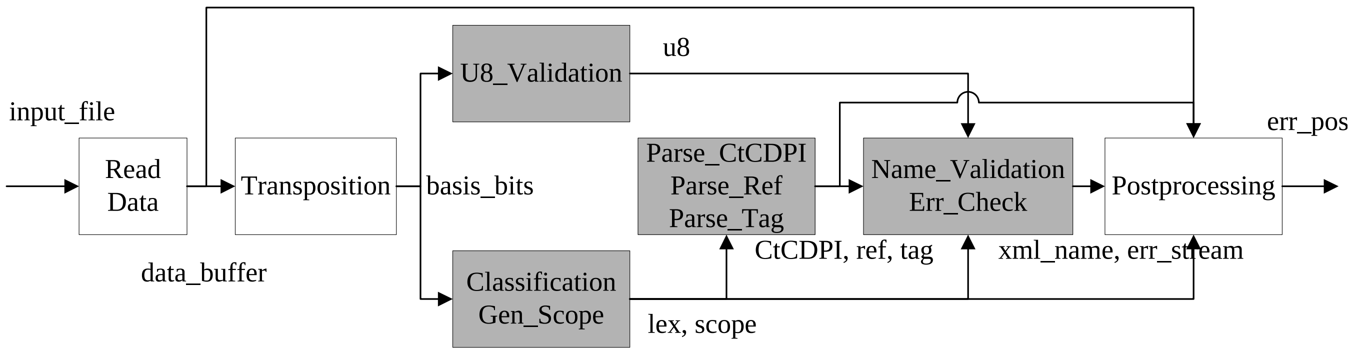


Figure 7: Parabix XML Parser Structure

`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 run-time library, as well as the hand-written Postprocessing code that completes the well-formed checking.

5 Evaluation Framework

XML Parsers: We evaluate the Parabix XML parser described above against two widely available open-source parsers: Xerces-C [12] and Expat [8]. Each of the parsers is evaluated on the task of implementing the parsing and well-formedness validation 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 config files in mobile phones. A key predictor of the overall parsing performance of an XML file is its *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 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.

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

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, register resources, and the addition of domain specific instructions. Here we investigate SIMD extensions across three different generations of intel processors (hardware details in Table 2). 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 supported by Sandybridge.

We investigated the execution profiles of each XML parser using the performance counters found in the processor. We chose several key hardware events that provide insight into the profile of each application and indicate if the processor is doing useful work [2, 3]. 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 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

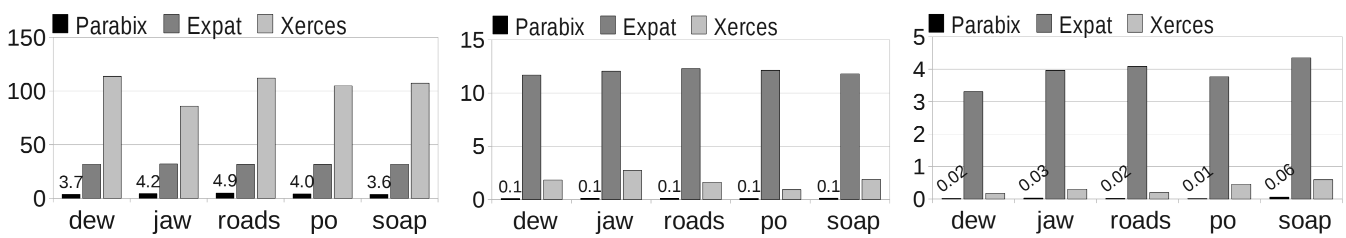
Energy Measurement: A key benefit of the Parabix parser is its more efficient use of the processor pipeline which reflects in the overall energy usage. We measure the energy consumption of the processor directly using a current clamp. We apply the Fluke i410 current clamp [9] to the 12V wires that supply 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 $12V * \sum_{i=1}^{N_{samples}} Sample_i$.

6 Efficiency of the Parabix-XML Parser

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 XML parsers on the Core-i3.

6.1 Cache behavior

The approximate miss penalty on the 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; L3 (4MB) 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. Compared to 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.

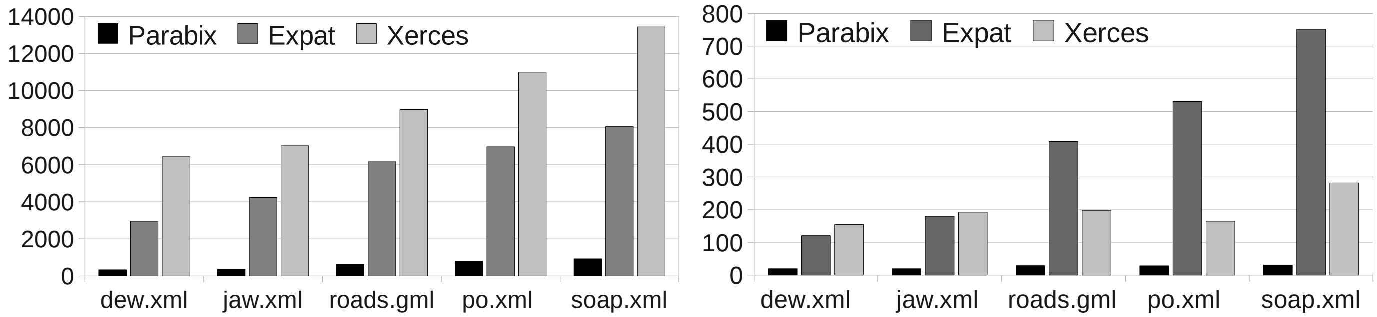


(a) L1 Misses (b) L2 Misses (c) L3 Misses

Figure 8: Cache Misses per kB of input data.

6.2 Branch Mispredictions

In general, performance is limited by branch mispredictions. Unfortunately, it is difficult to reduce the branch misprediction rate of traditional XML parsers due to: (1) the 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 1.0/1.1 specifications. As shown in Figure 9(a), Xerces averages up to 13 branches per XML byte processed on high density 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 is a significant overhead. In Parabix-XML,



(a) Branch Instructions / kB

(b) Branch Misses / kB

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

the use of SIMD operations eliminates many 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.

As shown in Figure 9(a), Parabix-XML is nearly branch free and exhibits minimal dependence on the source markup density. Specifically, it 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 9(b)) — which exceeds the average latency of a byte processed by Parabix-XML.

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 10 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.

6.4 CPU Cycles

Figure 11 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 before processing it; this requires several cycles per byte. However, transcoding using parallel bit streams is significantly faster and requires less than a single cycle per byte.

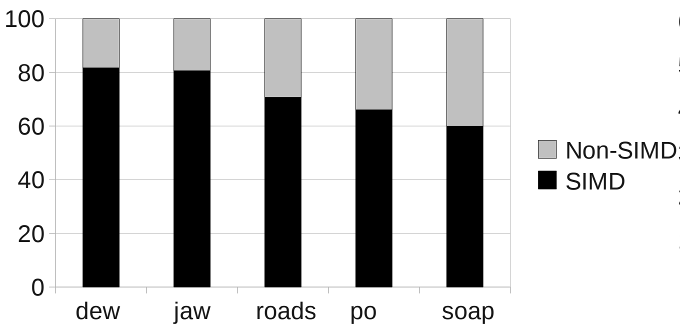


Figure 10: SIMD Instruction Percentage

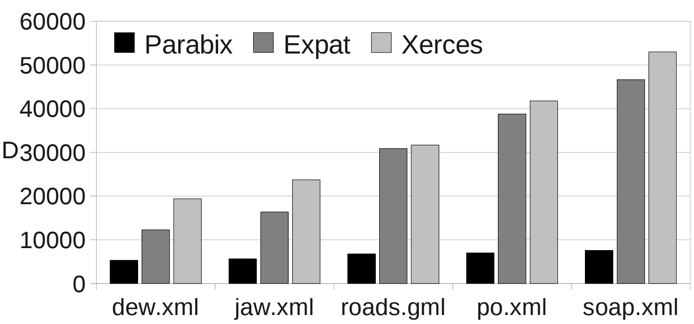
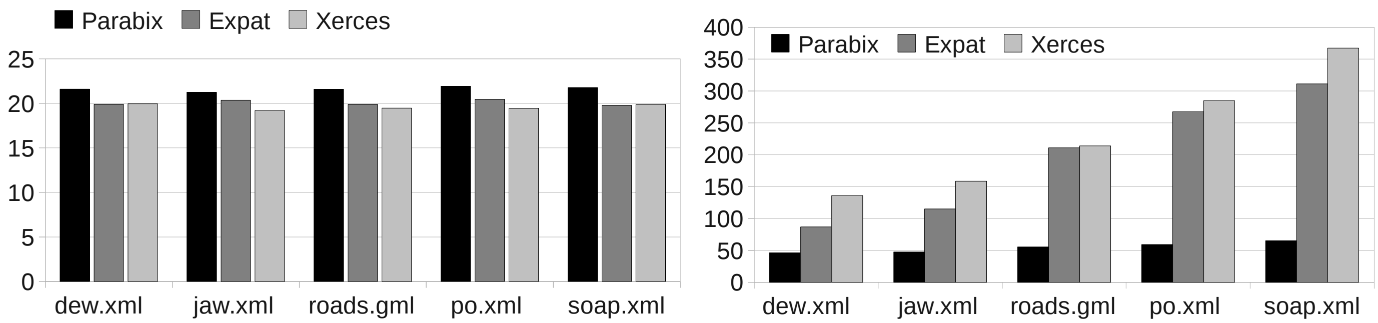


Figure 11: Performance (CPU Cycles per kB)

6.5 Power and Energy

In this section, we study the power and energy consumption of Parabix-XML in comparison with Expat and Xerces on Core-i3. Figure 12(a) shows the average power consumed by each parser. Parabix-XML, dominated by SIMD instructions, uses $\sim 5\%$ 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 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.



(a) Avg. Power (Watts) (b) Energy Consumption (μJ per kB)

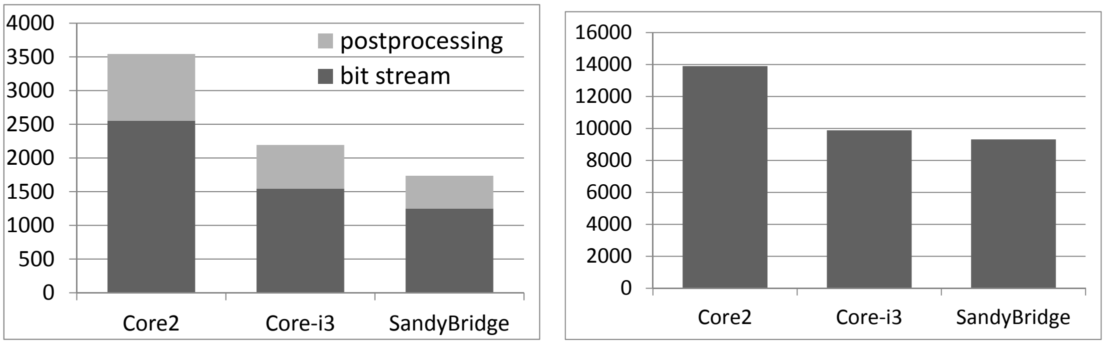
Figure 12: Power profile of Parabix on Core-i3

7 Evaluation of Parabix across different Hardware

7.1 Performance

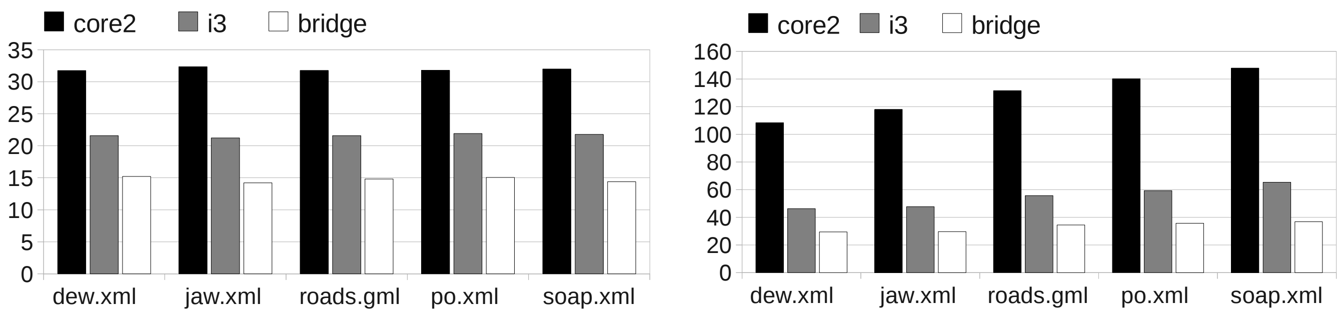
In this section, we study the performance of the XML parsers across three generations of Intel architectures. Figure 13(a) shows the average execution time of Parabix-XML (over all workloads). We analyze the execution time in terms of SIMD operations that operate on “bit streams” (*bit-space*) and scalar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

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 to be independent of the markup density of the input file. 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. For the purpose of comparison, Figure 13(b) shows the performance of the Expat parser. 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 the clock frequency and microarchitecture improvements while SandyBridge’s gains can be mainly attributed to the architecture. Figure 14(a) shows the average power consumption of Parabix-XML over each workload and as executed on each of the processor cores: Core2, Core-i3 and SandyBridge. Each generation of processor seem to bring with them 25–30% improvement in power consumption over the previous generation. Parabix-XML on SandyBridge consumes 72%–75% less energy than it did on Core2.



(a) Parabix (b) Expat

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



(a) Avg. Power of Parabix on various hardware (Watts) (b) Avg. Energy Consumption on various hardware (nJ per kB)

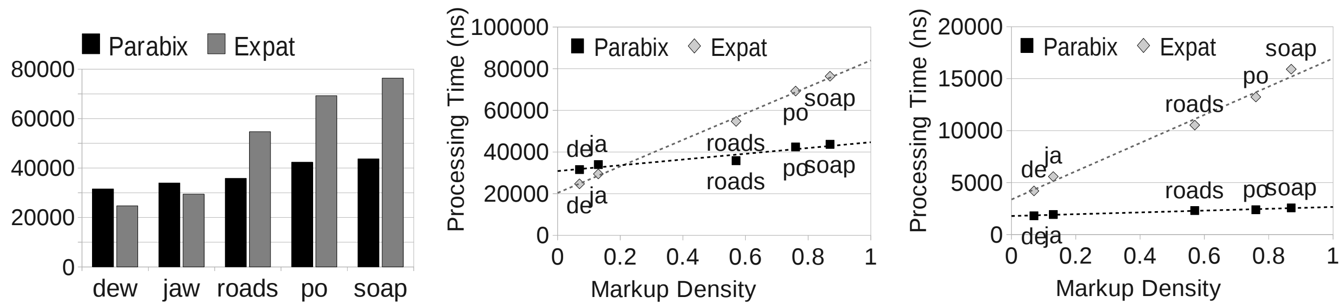
Figure 14: Energy Profile of Parabix on various hardware platforms

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 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. It 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 run-time 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 equivalent of those instructions was emulated using the available ISA. The resulting application was cross-compiled for Android using the Android NDK.

A comparison of Figure 15(a) and Figure 11 demonstrates that the performance of both Parabix and Expat degrades 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.



(a) ARM Neon Performance (cycles per kB)

(b) ARM Neon

(c) Core i3

Figure 15: Comparison of Parabix-XML on ARM vs. Intel.

Figure 15(b) investigates the performance of Expat and Parabix for the various input workloads on the Cortex-A8; Figure 15(c) plots the performance for Core-i3. The results demonstrate that 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 15(b) and 15(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 enhancement to ARM Neon that implement the Neon within the core microarchitecture could substantially improve the efficiency of Parabix-XML.

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 run-time 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, 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 or reload 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. AVX technology makes available the 3-operand form for both the new 256-bit operations as well as the base 128-bit SSE operations.

8.2 256-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 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.

8.3 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 run-time library to leverage the 256-bit AVX instructions 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 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. 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.

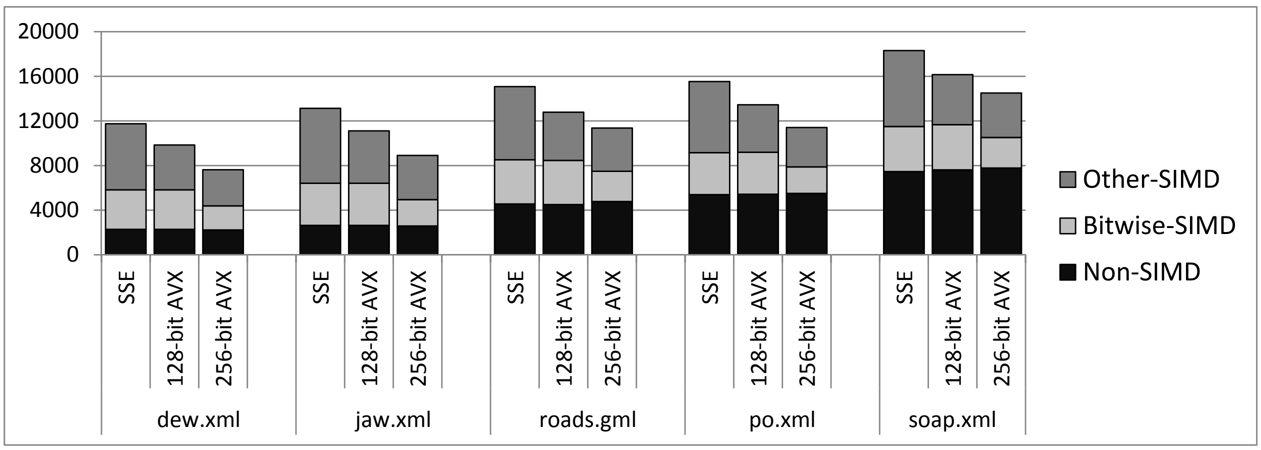


Figure 16: Parabix Instruction Counts (y-axis: Instructions per kB)

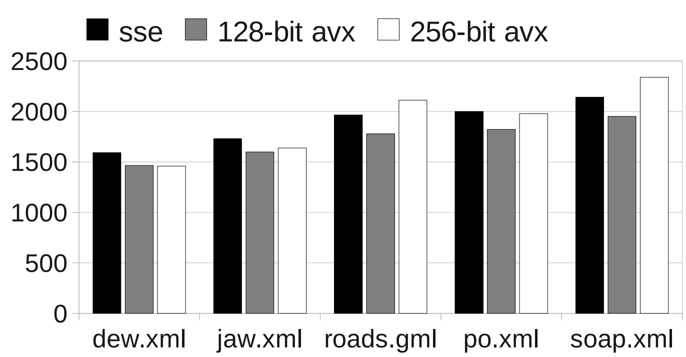


Figure 17: Parabix Performance (y-axis: ns per kB)

Note that, in each workload, the number of non-SIMD instructions remains relatively constant with each workload. As expected, the number of bitwise SIMD operations remains the same for both SSE and

128-bit 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 quite dramatic with the AVX extensions in Parabix demonstrating the ability of our runtime framework to exploit the available hardware resources. As shown in Figure 17, the benefits of the reduced SIMD instruction count are achieved only in the AVX 128-bit version. In this case, the benefits of 3-operand form seem to fully translate to performance benefits. Based on the reduction of overall Bitwise-SIMD instructions we expected a 11% improvement in performance. Instead, perhaps bizarrely, 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 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 in Parabix-XML.

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. However, up to this point, we restricted Parabix-XML to a single core. 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.

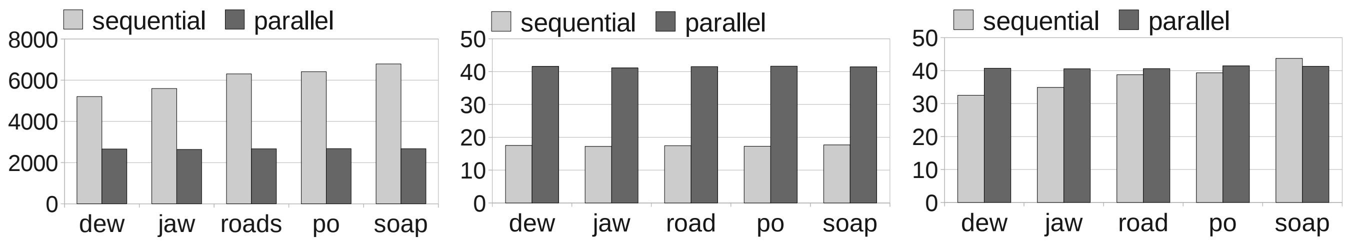
The typical approach to handling data parallelism with multiple threads involves partitioning data uniformly across the threads. However XML parsing is inherently sequential, which makes it difficult to partition the data. Several attempts have been made to address this problem using a preparsing phase to help determine the tree structure and to partition the XML document accordingly [18]. Another approach involved speculatively partitioning the data [21] but this introduced a significant level of complexity into the overall logic of the program.

			Data Structure Flow / Dependencies									
			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	write									
		transposition	read	write								
		classification		read		write						
Stage2	1.22	validate_u8		read	write							
		gen_scope				read	write					
		parse_CtCDPI				read	read	write				write
		parse_ref				read	read	read	write			
Stage3	2.03	parse_tag				read	read	read		write		
		validate_name			read	read		read	read	read	write	write
		gen_check			read	read	read	read		read	read	write
Stage4	1.32	postprocessing	read			read		read	read			read

Table 3: Relationship between Each Pass and Data Structures

In contrast to those methods, we adopted a parallelism strategy that requires neither speculation nor pre-parsing. As described in Section 4, Parabix-XML consists of multiple passes that, on every chunk of input data, 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. By analyzing the latency and data dependencies of each of the passes in the single-threaded version of Parabix-XML (Column 1 in Table 3), and assigned the passes to stages such that that provided the 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



(a) Performance (Cycles / kB) (b) Avg. Power Consumption (c) Avg. Energy Consumption (nJ / Byte)

Figure 18: Multithreaded Parabix

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.

Figure 18 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 SIMD units of all SandyBridge's cores. This performance approaches the 1 cycle per byte performance of custom hardware solutions [10]. 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 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 (see Figure 18(c)) being similar to the the single-thread execution (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 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 [19]. 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 [17, 20] and speculative p-DFAs [24]. Hardware methods include custom XML chips [15] and FPGA-based implementations [10]. Intel's SSE4 instructions targeted XML parsers, but these have not seen widespread use because of portability concerns and the programming challenges that accompany low level instructions [14]. Recently, Cameron et al. [6,7] designed an accelerated XML parser using widely available SSE2 instructions. Finally, others have explored the design of custom hardware for bit parallel operations for text search in network processors [22].

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

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

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

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

[4] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, 2010.

[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, 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.

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

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

[9] F. Corporation. Fluke Clamp Meters. http://www.fluke.com/.

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

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, 2011.

[12] A. S. Foundation. Xerces C++ Parser. http://xerces.apache.org/xerces-c/.

[13] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, 2010.

[14] Z. Lei. Xml parsing accelerator with intel streaming simd extensions 4. http://software.intel.com/en-us/articles/xml-parsing-accelerator-with-intel-streaming-simd-extensions-4-intel-sse4/.

[15] 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.

[16] M. Leventhal and E. Lemoine. The xml chip at 6 years. In *In Proceedings of the International Symposium on Processing XML Efficiently.*, Aug 2009.

[17] X. Li, H. Wang, T. Liu, and W. Li. Key elements tracing method for parallel XML parsing in multi-core system. *Parallel and Distributed Computing Applications and Technologies, International Conference on*, 0:439–444, 2009.

[18] W. Lu, Y. Pan, , and K. Chiu. A parallel approach to xml parsing. *The 7th IEEE/ACM International Conference on Grid Computing*, 2006.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.