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

Paper ID ****

Abstract

In modern applications text files are employed widely. For example, XML files provide data storage in human readable format and are widely used in web services, database systems, and 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 processed character and experience high cache miss rates when parsing the file. Overall, parsing text in important domains like XML processing requires high performance motivating hardware designers to adopt customized hardware and ASIC solutions.

In this paper, we enable text processing applications to effectively use commodity processors. We introduce Parabix (Parallel Bitstream) technology, a software runtime and execution model that allows applications to exploit modern SIMD instructions extensions for high performance text processing. Parabix enables the application developer to write constructs assuming unlimited SIMD data parallelism. Our runtime translator generates code based on machine specifics (e.g., SIMD register widths) to realize the programmer specifications. The key insight into efficient text processing in Parabix is the data organization. It transposes the sequence of 8-bit characters into sets of 8 parallel bit streams which then enables us to operate on multiple characters with single bit-parallel SIMD operators. We demonstrate the features and efficiency of parabix with a XML parsing application. We evaluate a Parabix-based XML parser against two widely used XML parsers, Expat and Apache's Xerces, and across three generations of x86 processors, including the new Intel SandyBridge. We show that Parabix's speedup is 2×–7× over Expat and Xerces. We observe that Parabix overall makes efficient use of intra-core parallel hardware on commodity processors and supports significant gains in energy. Using Parabix, we assess the scalability advantages of SIMD processor improvements across Intel processor generations, culminating with a look at the latex 256-bit AVX technology in SandyBridge versus the now legacy 128-bit SSE technology. As part of this study we also preview the Neon extensions on ARM processors. Finally, we partition the XML program into pipeline stages and demonstrate that thread-level parallelism exploits SIMD units scattered across the different cores and improves performance (2× on 4 cores) at same energy levels as the single-thread version.

1 Introduction

We have now long since reached the limit to classical Dennard voltage scaling that enabled us to keep all of transistors afforded by Moore's law active. This has already resulted in a rethink of the way general-

purpose processors are built: processor frequencies have remained stagnant over the last 5 years with the capability to boost core speeds on Intel multicores 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 [5], it is not certain how many cores can be productively used in scaling our chips [14]. This is because exploiting parallelism across multiple cores tends to require heavweight threads that are difficult to manage and synchronize.

The desire to improve the overall efficiency of computing is pushing designers to explore customized hardware [18, 22] 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] and show how Parabix, a novel software architecture, tool chain and run-time environment can indeed be used to dramatically improve parsing efficiency on commodity processors. Based on the concept of transposing byte-oriented character data into parallel bit streams for the individual bits of each byte, the Parabix framework 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 achieve transposition, Parabix exploits sophisticated SIMD instructions that enable data elements to be packed and unpacked from registers in a regular manner which improves the overall cache access behavior of the application resulting in significantly fewer misses and better utilization. Parabix also dramatically reduces branches in parsing code resulting in a more efficient pipeline and substantially improves register/cache utilization which minimizes energy wasted on data transfers.

We apply Parabix technology to the problem of XML parsing and develop several implementations for

different computing platforms. XML is a particularly interesting application; it 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, as well as ubiquitous XML data in Android phones. XML parsing efficiency is important for multiple application areas; in server workloads the key focus in on overall transactions per second, while in applications in network switches and cell phones, latency and energy are of paramount importance. Traditional 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 memory bandwidth and data caches due to byte-at-a-time processing and multiple buffering. XML ASIC chips have been around for over 6 years, but typically lag behind CPUs in technology due to cost constraints. Our focus is how much we can improve performance of the XML parser on commodity processors with Parabix technology.

In the end, as summarized by Figure 1 our Parabix-based XML parser improves the performance by and energy efficiency several-fold compared to widely-used software parsers and approaching the performance of performance of ASIC XML parsers.[1]

Overall we make the following contributions in this paper.

1) We introduce the Parabix architecture, tool chain and run-time 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.

2) We compare our Parabix XML parsers against conventional parsers and assess the improvement in overall performance and energy efficiency on each platform. 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, possibly due to their memory interface. We compare the ARM Neon extensions against the x86 SIMD extensions and comment on the latency of SIMD operations across these architectures.

3) Finally, building on the SIMD parallelism of Parabix technology, we multithread the Parabix XML

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

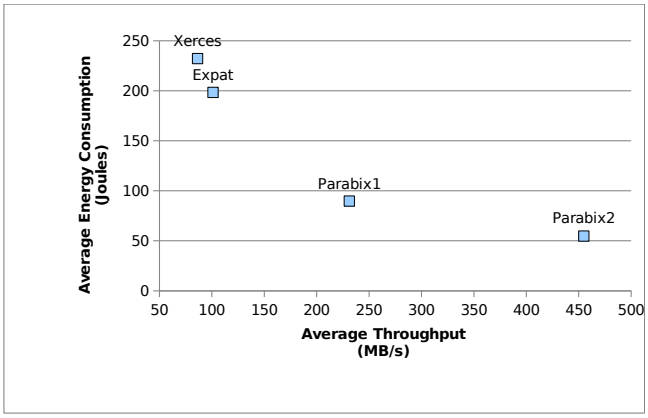


Figure 1: XML Parser Technology Energy vs. Performance

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 on mainstream processors. Section 3 describes the Parabix architecture, tool chain and run-time environment. Section 4 describes the application of the Parabix framework to the construction of an XML parser meeting enforcing all the well-formedness rules of the XML specification. Section 5 then describes the overall methodology of our performance and energy study. Section 6 presents a detailed performance evaluation on a Core-i3 processor as our primary evaluation platform, addressing a number of microarchitectural issues including cache misses, branch mispredictions, and SIMD instruction counts. Section 7 examines scalability and performance gains through three generations of Intel architecture. Section 8 examines the extension of the Parabix technology to take advantage of Intel's new 256-bit AVX technology, while Section 9 investigates the applications of this technology on mobile platforms using ARM processors with Neon SIMD extensions. Section 10 then looks at the multithreading of the Parabix XML parser using pipeline parallelism. Section 11 concludes the paper.

2 Background

2.1 XML

In 1998, the W3C officially adopted XML as a standard. The defining characteristics of XML are that it can represent virtually any type of information through the use of self-describing markup tags and can easily store semi-structured data in a descriptive fashion. XML markup encodes a description of an XML document's storage layout and logical structure. Because XML was intended to be human-readable, XML markup tags are often verbose by design [6].

XML files can be classified as "document-oriented" or "data-oriented" [13]. Documented-oriented XML is designed for human readability, such as shown in Figure 2; data-oriented XML files are intended to be parsed by machines and omit "human-friendly" formatting techniques, such as the use of whitespace and descriptive "natural language" naming schemes. Although the XML specification itself does not distinguish between "XML for documents" and "XML for data" [6], the latter often requires the use of an XML parser to extract the information within. The role of an XML parser is to transform the text-based XML data into application ready data.

```
<?xml version="1.0"?>
<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: Example XML Document

2.2 Traditional XML Parsers

Traditional XML parsers process XML sequentially a single byte-at-a-time. Following this approach, an XML parser processes a source document serially, from the first to the last byte of the source file. Each character of the source text is examined in turn to distinguish between the XML-specific markup, such as an opening angle bracket '¡', and the content held within the document. The current character that the parser is processing is commonly referred to using the concept of a current cursor position. As the parser moves the cursor through the source document, the parser alternates between markup scanning, and data validation and processing operations. At each processing step, the parser scans the source document

and either locates the expected markup, or reports an error condition and terminates. In other words, traditional XML parsers operate as complex finite-state machines that use byte comparisons to transition between data and metadata states. Each state transition indicates the context in which to interpret the subsequent characters. Unfortunately, textual data tends to consist of variable-length items sequenced in generally unpredictable patterns [9]; thus any character could be a state transition until deemed otherwise.

Expat and Xerces-C are popular byte-a-time sequential parsers. Both are C/C++ based and open-source. Expat was originally released in 1998; it is currently used in Mozilla Firefox and provides the core functionality of many additional XML processing tools [11]. Xerces-C was released in 1999 and is the foundation of the Apache XML project [17].

A major disadvantage of the sequential byte-at-a-time approach to XML parsing is that each XML character incurs at least one conditional branch. The cumulative effect of branch mispredictions penalties are known to degrade XML parsing performance in proportion to the markup density of the source document [10] (i.e., the proportion of XML-markup to XML-data).

2.3 Parallel XML Parsing

In general, parallel XML acceleration methods come in one of two forms: multithreaded approaches and SIMD-based techniques. Multithreaded XML parsers take advantage of multiple cores via number of strategies. Common strategies include preparsing the XML file to locate key partitioning points [23] and speculative p-DFAs [23]. SIMD XML parsers leverage the SIMD registers to overcome the performance limitations of the sequential byte-at-a-time processing model and its inherently data dependent branch misprediction rates. Further, data parallel SIMD instructions allow the processor to perform the same operation on multiple pieces of data simultaneously. The Parabix1 and Parabix2 parsers studied in this paper fall under the SIMD classification. The Parabix parser versions studied are described in further detail in Section 3.

3 Parabix

This section provides an overview of the SIMD-based parallel bit stream XML parsers, Parabix1 and Parabix2. A comprehensive study of Parabix2 can be found in the technical report “Parallel Parsing with Bitstream Addition: An XML Case Study” [9].

3.1 Parabix1

Parabix1 processes source XML in a functionally equivalent manner as a traditional recursive descent XML parser. That is, Parabix1 moves sequentially through the source document, maintains a single parser cursor position, and parsers recursively and depth-first. Where Parabix1 differs from the traditional parser is that it scans for key markup characters using a series of bit streams. A bit stream is simply a sequence of 0s and 1s. A 1-bit marks the postion of each key character in the corresponding source data stream. A single stream is generated for each of the key markup characters.

In Parabix1, basis bit streams are used to generate character-class streams for key markup characters. Basis bit streams are defined as the set of bit streams that represent the transposed data format of the source XML byte data. In other words, M-bit source characters are represented in transposed representation using M basis bit streams. Figure 3 presents an example of the basis bit stream representation of 8-bit ASCII characters. $B_0 \ldots B_7$ are the individual bit streams. The 0 bits in the bit streams are represented by periods as to emphasize the 1 bits.

source data	`<t1>abc</t1><tag2/>`
B_0	`..1.1.1.1.1...11.1.`
B_1	`...1.11.1..1...1111`
B_2	`11.1...111.111.1.11`
B_3	`1..1...11..11....11`
B_4	`1111...1.11111..1.1`
B_5	`1111111111111111111`
B_6	`.1..111..1...111...`
B_7	`...................`

Figure 3: Example 8-bit ASCII Character Basis Bit Streams

To transform byte-oriented character data to parallel bit stream representation, source data is first loaded into SIMD registered in sequential order. It is then converted to the transposed basis bit stream representation through a series of parallel SIMD pack, shift, and logical bitwise operations. Using the SIMD capabilities of current commodity processors, the transposition of source data to basis bit stream format incurs an amortized cost of approximately 1 cycle per byte [10].

Throughout the XML parsing process we must identify key XML characters. For example, the opening angle bracket character '¡'. For this purpose, we combine the basis bit streams using bitwise logic and

compute character-class bit streams. For example, the j-th character is an open angle bracket '¡' if and only if the j-th bit of $B_2, B_3, B_4, B_5 = 1$ and the j-th bit of $B_0, B_1, B_6, B_7 = 0$. Character-class streams mark the positions of source characters as a single 1-bit. Each bit position in the computed bit stream is in one-to-one correspondence with its source byte position. Once generated, single cycle built-in *bitscan* operations are used to locate the positions of key XML characters throughout the parsing process. Utilizing M SIMD registers of width W, it is possible to scan through W characters in parallel. The register width W is processor dependent and ranges from 64-bit for MMX, to 128-bit for SSE, and 256-bit for AVX.

A common operation in XML parsing is XML start tag validation. Starts tags begin with '¡' and end with either "/¿" or "¿" (depending on whether the element tag is an empty element tag or not, respectively). Figure 4 conceptually demonstrates start tag validation as performed in Parabix1 using character-class streams together with the processor built-in *bitscan* operation. We proceeed as follows. The first bit stream M_0 is created and the parser begins scanning the source data for an open angle bracket '¡', starting at position 1. Since the source data begins with '¡', M_0 is assigned a cursor position of 1. The *advance* operation then shifts M_0's cursor position by 1, resulting in the creation of a new bit stream, M_1, with the cursor position at 2. The following *bitscan* operation takes the cursor position from M_1 and sequentially scans every position until it locates either an '¿'. It finds a '¿' at position 4 and returns that as the new cursor position for M_2. Calculating M_3 advances the cursor again, and the *bitscan* used to create M_4 locates the new opening angle bracket. This process continues in sequence until until all start tags are validated. Unlike traditional parsers, these sequential operations are accelerated significantly since the *bitscan* operation can skip up to w positions, where w is the processor word width in bits. This approach has recently been applied to Unicode transcoding and XML parsing to good effect, with research prototypes showing substantial speed-ups over even the best of byte-at-a-time alternatives [7, 10, 19].

3.2 Parabix2

In Parabix2, the sequential single-cursor parsing approach using *bitscan* instructions is replaced by a parallel parsing approach, that uses multiple cursors when possible, and bit stream addition operations to advance multiple cursor positions in parallel. Unlike the single-cursor approach of Parabix1 (and conceptually of all sequential XML parsers), Parabix2 processes multiple cursors in parallel. For example, using

source data	`<t1>abc</t1><tag2/>`
$M_0 = 1$	`1...............`
$M_1 = advance(M_0)$	`.1..............`
$M_2 = bitscan('>')$	`...1............`
$M_3 = advance(M_2)$	`....1...........`
$M_4 = bitscan('<')$	`.......1.......`
$M_5 = advance(M_4)$	`........1......`
$M_6 = advance(M_5)$	`.........1.....`
$M_7 = bitscan('<')$	`...........1....`
$M_8 = advance(M_7)$	`............1...`
$M_9 = bitscan('/')$	`...............1.`
$M_{10} = advance(M_9)$	`................1`

Figure 4: Parabix1 Start Tag Validation

the source data from Figure 4, Figure 5 conceptually demonstrates the manner in which Parabix2 identifies and advances each of the start tag bit streams. Unlike Parabix1, Parabix2 begins scanning by creating two character-class bit streams, N, denoting the position of every alpha numeric character within the basis stream, and M_0, marking the position of every potential start tag in the bit stream. M_0 is advanced to create M_1, which is fed into the first *scanto* operation along with N. To handle variable length tag names, the *scanto* operation effectively locates the cursor positions of the end tags in parallel by adding M_1 to N, and uses the bitwise AND operation of the negation of N to find only the true end tags of M_1. Because an end tag may end on an '/' or '¿', *scanto* is called again to advance any cursor from '/' to '¿'. For additional details, refer to the technical report [9].

source data	`<t1>abc</t1><tag2/>`
$N =$ Tag Names	`.11.....11..1111..`
$M_0 =$ `[<]`	`1..........1.....`
$M_1 = advance(M_0)$	`.1..........1....`
$M_2 = scanthru(M_1, A)$	`...1............1.`

Figure 5: Parabix2 Start Tag Validation

In general, the set of bit positions in a bit stream may be considered to be the current parsing positions of multiple parses taking place in parallel throughout the source data stream. Although it is not explicitly shown in these prior examples, error bit streams can be used to identify any well-formedness errors found

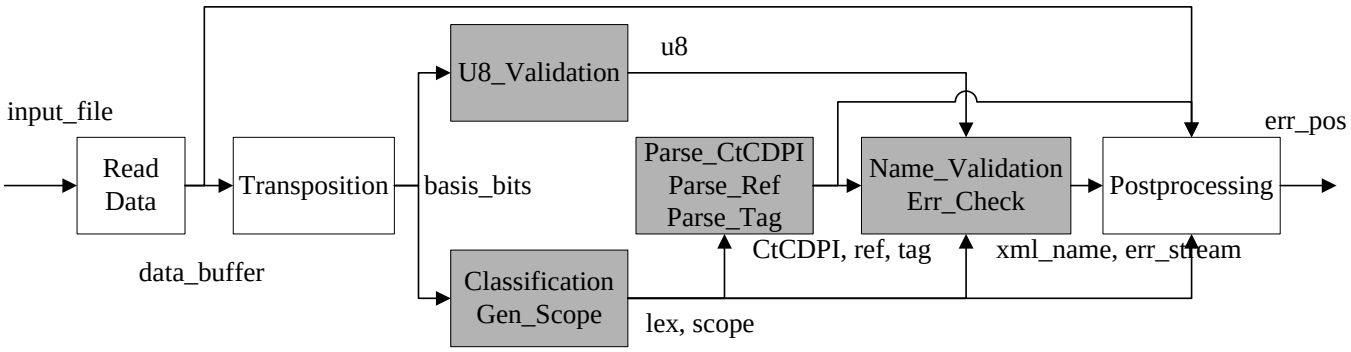


Figure 6: Parabix2 Structure

during the parsing process. Error positions are gathered and processed in as a final post processing step. A further aspect of the parallel cursor method with bit stream addition is that the conditional branch statements used to identify syntax error at each each parsing position are eliminated. Hence, Parabix2 offers additional parallelism over Parabix1 in the form of multiple cursor parsing and further reduces branch misprediction penalties.

4 The Parabix XML Parser

4.1 Parser Structure

Figure 6 shows the overall structure of the Parabix XML parser set up for well-formedness checking. The input file is processed using 11 functions organized into 7 modules. In the first module, the Read_Data function loads data blocks from an input file to data_buffer. The data is then transposed to eight parallel basis bitstreams (basis_bits) in the Transposition module. The eight bitstreams are used in the Classification function to generate all the XML lexical item streams (lex) as well as in the U8_Validation module to validate UTF-8 characters. The lexical item streams and scope streams (scope) that are generated in Gen_Scope function are supplied to the parsing module, which consists three functions, Parse_CtCDPI, Parse_Ref and Parse_tag. These functions deal with the parsing of comments, CDATA sections, processing instructions, references and tags. After this, information is gathered by Name_Validation and Err_Check functions, producing name check streams and error streams. These are then passed to the final module for Postprocessing. All the possible errors that cannot be conveniently detected by bitstreams are checked in this last module. The final output reports any well-formedness error detected and its position within the

input file.

4.2 Parallel Bit Stream Compilation

While the description of parallel bit stream parsing in the previous section works conceptually on unbounded bit streams, in practice, a corresponding C implementation to process input streams into blocks of size equal to the SIMD register width of the target processor is required. In our work, we leverage the unbounded integer type of the Python programming language. Using a restricted subset of Python, we prototype and validate the functionality of applications, such as XML validation and UTF-8 to UTF-16 transcoding. We then compile this Python code into equivalent block-at-a-time C code. The key question becomes how to transfer information from one block to the next whenever token scans cross block boundaries.

The answer lies in carry bit propagation. Since the parallel *scanto* operation relies solely on bit-wise addition and logical operations, block-to-block information transfer is captured in entirety by the carry bit associated with each underlying addition operation. Logical operations do not require information flow across block boundaries. Properly determining, initializing and inserting carry bits into a block-by-block implementation is tedious and error prone. Thus we have developed compiler technology to automatically transform parallel bit stream Python code to block-at-a-time C implementations. Details are beyond the scope of this paper, but are described in the on-line source code repository at parabix.costar.sfu.ca.

5 Methodology

In this section we describe our methodology for the measurements and investigation of XML parser energy consumption and performance. In brief, for each of the four XML parsers under study we propose to measure and evaluate the energy consumption required to carry out XML well-formedness checking, under a variety of workloads, and as executed on three different Intel processors.

To begin our study we propose to first investigate each of the XML parsers in terms of the Performance Monitoring Counter [2] (PMC) hardware events listed in the PMC Hardware Events subsection. Based on

[2]Performance Monitoring Counters are special-purpose registers available with most modern microprocessors. PMCs store the running count of specific hardware events, such as retired instructions, cache misses, branch mispredictions, and arithmetic-logic unit operations. PMCs can be used to capture information about any program at run-time and under any workload at a fine granularity.

the findings of previous work [2–4] we have chosen several key hardware performance events for which the authors indicate a strong correlation with energy consumption. In addition, we measure the runtime counts of SIMD instructions and bitwise operations using the Intel Pin binary instrumentation framework. Based on these data we gain further insight into XML parser execution characteristics and compare and constrast each of the Parabix parser versions against the performance of standard industry parsers.

The foundational work by Bellosa in [2] as well as more recent work in [3,4] demonstrate that hardware-usage patterns have a significant impact on the energy consumption characteristics of an application [2–4]. Further, the authors demonstrate a strong correlation between specific PMC events and energy usage. However, each author differs slightly in their opinion of the exact set of PMCs to use.

The following subsections describe the XML parsers under study, XML workloads, the hardware architectures, PMC hardware events selected for measurement, and the energy measurement instrumentation set up. We analyze the performance of each of the XML parsers under study based on PMC hardware event counts and contrast their energy consumption measurements based on direct measurements.

5.1 Parsers

The XML parsing technologies selected for this study are the Parabix1, Parabix2, Xerces-C++, and Expat XML parsers. Parabix1 (parallel bit Streams for XML) is our first generation SIMD and Parallel Bit Stream technology based XML parser [15]. Parabix1 leverages the processor built-in *bitscan* operation for high-performance XML character scanning as well as the SIMD capabilities of modern commodity processors to achieve high performance. Parabix2 [16] represents the second generation of the Parabix1 parser. Parabix2 is an open-source XML parser that also leverages Parallel Bit Stream technology and the SIMD capabilities of modern commodity processors. However, Parabix2 differs from Parabix1 in that it employs new parallelization techniques, such as a multiple cursor approach to parallel parsing together with bit stream addition techniques to advance multiple cursors independently and in parallel. Parabix2 delivers dramatic performance improvements over traditional byte-at-a-time parsing technology. Xerces-C++ version 3.1.1 (SAX) [17] is a validating open source XML parser written in C++ by the Apache project. Expat version 2.0.1 [11] is a non-validating XML parser library written in C.

File Name	dewiki.xml	jawiki.xml	roads.gml	po.xml	soap.xml
File Type	document	document	data	data	data
File Size (kB)	66240	7343	11584	76450	2717
Markup Item Count	406792	74882	280724	4634110	18004
Markup Density	0.07	0.13	0.57	0.76	0.87

Table 1: XML Document Characteristics

5.2 Workloads

Markup density is defined as the ratio of the total markup contained within an XML file to the total XML document size. This metric has substantial influence on the performance of traditional recursive descent XML parser implementations. We use a mixture of document-oriented and data-oriented XML files in our study to provide 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 7-bit encoded ASCII characters.

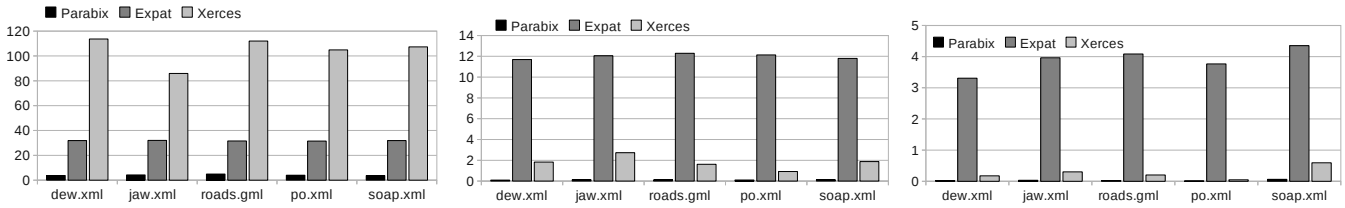
5.3 Platform Hardware

Intel Core2 Intel Core2 processor, code name Conroe, produced by Intel. Table **??** gives the hardware description of the Intel Core2 machine.

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

Intel Core-i3 processor, code name Nehalem, produced by Intel. The intent of the selection of this processor is to serve as an example of a low end server processor. Table **??** gives the hardware description of the Intel Core-i3 machine. Intel Core-i5 processor, code name SandyBridge produced by Intel. Table **??** gives the hardware description of the Intel Core-i3 machine. Each of the hardware events selected relates to performance and energy features associated with one or more hardware units. For example, total



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

Figure 7: Cache Misses per kB of input data.

branch mispredictions relate to the branch predictor and branch target buffer capacity.

The set of PMC events used included in this study are as follows. Processor Cycles, Branch Instructions, Branch Mispredictions, Integer Instructions, SIMD Instructions and Cache Misses.

5.4 Energy Measurement

We measure energy consumption using the Fluke i410 current clamp applied on 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 multimeter at the granularity of 100 samples per second. This measurement captures the power to the processor package, including cores, caches, Northbridge memory controller, and the quick-path interconnects [12].

6 Baseline Evaluation on Core-i3

6.1 Cache behavior

Core-i3 has a three level cache hierarchy. The approximate miss penalty for each cache level is 4, 11, and 36 cycles respectively. Figure 7(a), Figure 7(b) and Figure **??** show the L1, L2 and L3 data cache misses for each of the parsers. Although XML parsing is non memory intensive application, cache misses for the Expat and Xerces parsers represent a 0.5 cycle per XML byte cost whereas the performance of the Parabix parsers remains essentially unaffected by data cache misses. Cache misses not only consume additional CPU cycles but increase application energy consumption. L1, L2, and L3 cache misses consume approximately 8.3nJ, 19nJ, and 40nJ respectively. As such, given a 1GB XML file as input, Expat and Xerces would consume over 0.6J and 0.9J respectively due to cache misses alone.

6.2 Branch Mispredictions

Despite improvements in branch prediction, branch misprediction penalties contribute significantly to XML parsing performance. On modern commodity processors the cost of a single branch misprediction is commonly cited as over 10 CPU cycles. As shown in Figure 8(b), the cost of branch mispredictions for the Expat parser can be over 7 cycles per XML byte—this cost alone is equal to the average total cost for Parabix2 to process each byte of XML.

In general, reducing the branch misprediction rate is difficult in text-based XML parsing applications. This is due in part to the variable length nature of the syntactic elements contained within XML documents, a data dependent characterstic, as well as the extensive set of syntax constraints imposed by the XML 1.0 specification. As such, traditional byte-at-a-time XML parsers generate a performance limiting number of branch mispredictions. As shown in Figure 8(a), Xerces averages up to 13 branches per XML byte processed on high density markup.

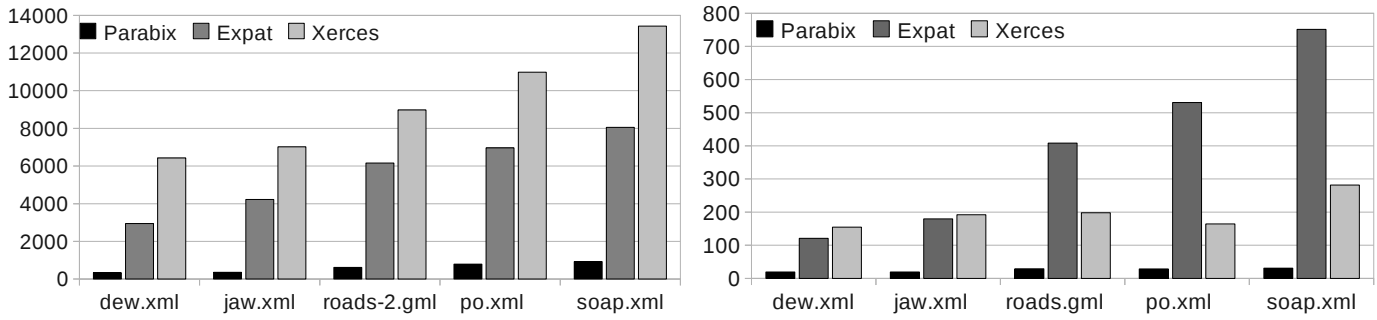
The performance improvement of Parabix1 in terms of branch mispredictions results from the veritable elimination of conditional branch instructions in scanning. Leveraging the processor built-in *bit scan* operation together with parallel bit stream technology Parabix1 can scan up to 64 bytes of source XML with a single *bit scan* instruction. In comparison, a byte-at-a-time parser must process a conditional branch instruction per XML byte scanned.

As shown in Figure 8(a), Parabix2 processing is almost branch free. Utilizing a new parallel scanning technique based on bit stream addition, Parabix2 exhibits minimal dependence on source XML markup density. Figure 8(a) displays this lack of data dependence via the constant number of branch mispredictions shown for each of the source XML files.

6.3 SIMD Instructions vs. Total Instructions

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

Using the Intel Pin tool, we gather the dynamic instruction mix for each XML workload, and classify



(a) Branch Instructions (b) Branch Misses

Figure 8: Branch characteristics on the Core-i3 per kB of input data.

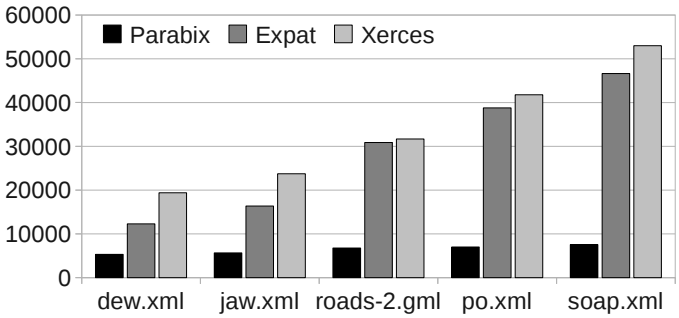
instructions as either vector (SIMD) or non-vector instructions. Figures **??** and 9(b) show the percentage of SIMD instructions for Parabix1 and Parabix2 respectively. For Parabix1, 18% to 40% of the executed instructions are SIMD instructions. Using bit stream addition to scan XML characters in parallel, the Parabix2 instruction mix is made up of 60% to 80% SIMD instructions. Although the resulting ratios are (negatively) proportional to the markup density for both Parabix1 and Parabix2, the degradation rate of Parabix2 is much lower and thus the performance penalty incurred by increasing the markup density is reduced.

6.4 CPU Cycles

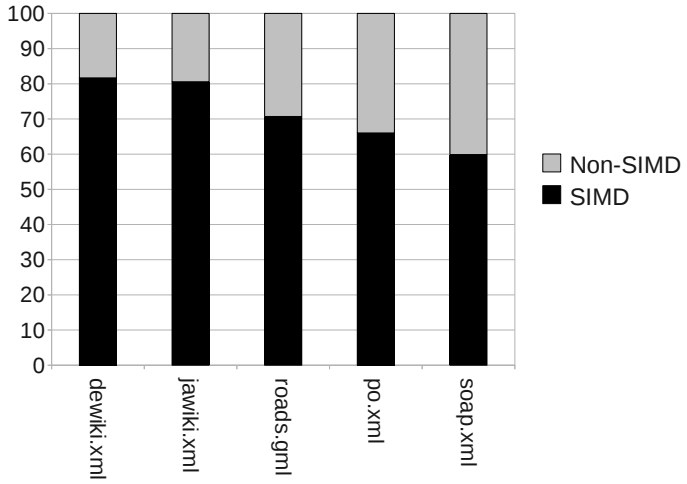
Figure 9(a) shows overall parser performance evaluated in terms of CPU cycles per kilobyte. Parabix1 is 1.5 to 2.5 times faster on document-oriented input and 2 to 3 times faster on data-oriented input than the Expat and Xerces parsers respectively. Parabix2 is 2.5 to 4 times faster on document-oriented input and 4.5 to 7 times faster on data-oriented input. Traditional parsers can be dramatically slowed by dense markup, while Parabix2 is generally unaffected. The results presented are not entirely fair to the Xerces parser since it first transcodes input from UTF-8 to UTF-16 before processing. In Xerces, this transcoding requires several cycles per byte. However, transcoding using parallel bit streams is significantly faster and requires less than a single cycle per byte. [8].

6.5 Power and Energy

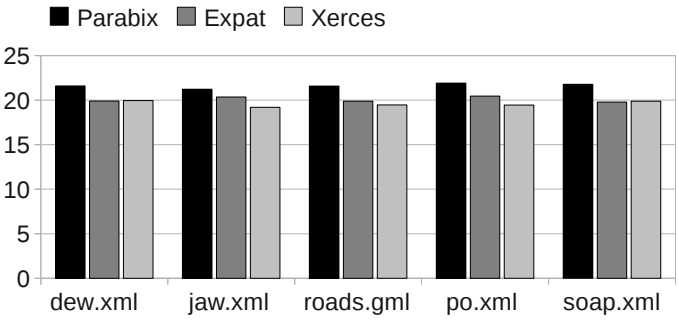
In response to the growing industry concerns on power consumption and energy efficiency, chip producers work hard to not only improve performance but also achieve high energy efficiency in processors design. We study the power and energy consumption of Parabix in comparison with Expat and Xerces on



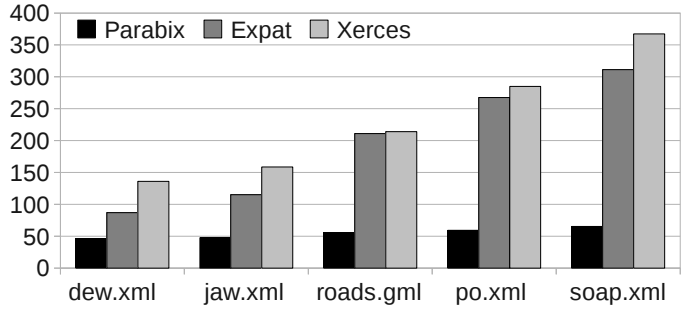
(a) Performance : # Cycles/kb



(b) SIMD Instruction Breakdown. Y Axis : % SIMD Instruction/kb



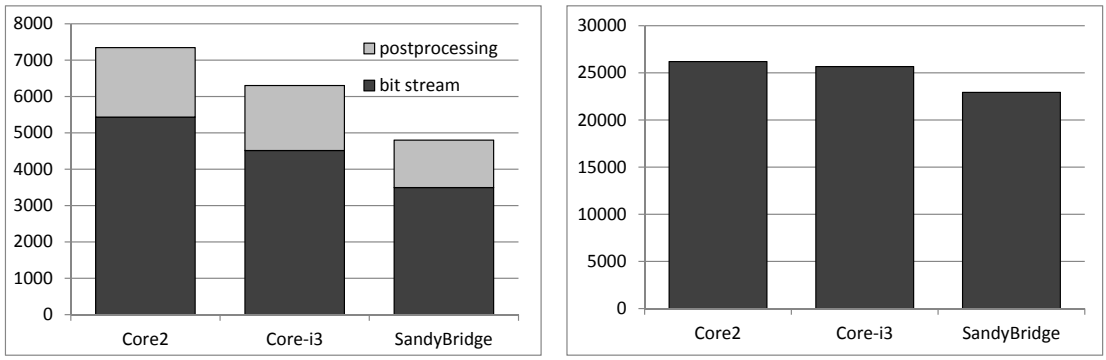
(c) Avg. Power (Watts)



(d) Energy Consumption (μJ per kB)

Core-i3. The average power of Core-i3 530 is about 21 watts. This Intel model has a good reputation for power efficiency. Figure 9(c) shows the average power consumed by each parser. Parabix2, dominated by SIMD instructions, uses approximately 5% additional power.

As shown in Figure 9(d), a comparison of energy efficiency demonstrates a more interesting result. Although Parabix2 requires slightly more power (per instruction), the processing time of Parabix2 is significantly lower, and therefore Parabix2 consumes substantially less energy than the other parsers. Parabix2 consumes 50 to 75 nJ per byte while Expat and Xerces consume 80nJ to 320nJ and 140nJ to 370nJ per byte respectively.



(e) Parabix2 (f) Expat

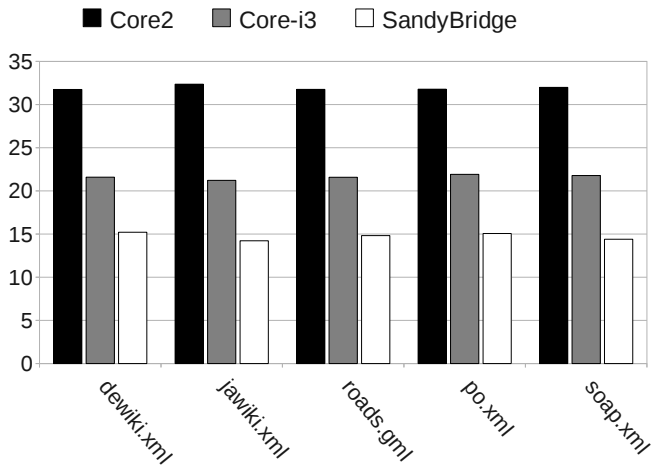
Figure 9: Average Performance Parabix vs. Expat (y-axis: CPU Cycles per kB)

7 Scalability

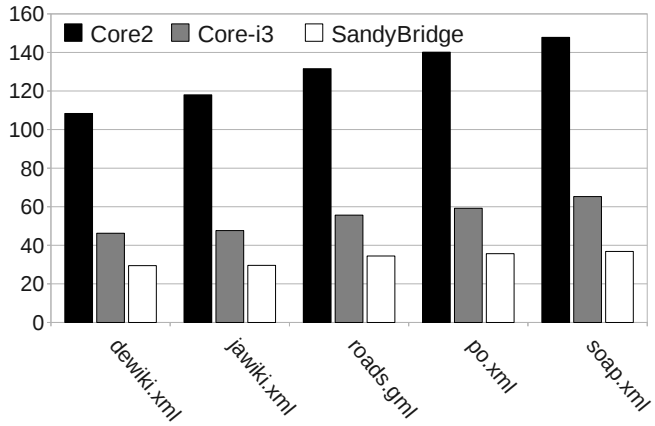
7.1 Performance

Figure 9 (a) demonstrates the average XML well-formedness checking performance of Parabix2 for each of the workloads and as executed on each of the processor cores — Core2 Core-i3 and SandyBridge. Processing time is shown in terms of bit stream based operations executed in 'bit-space' and postprocessing operations executed in 'byte-space'. In the Parabix2 parser, bit-space parallel bit stream parser operations consist primarily of SIMD instructions; byte-space operations consist of byte comparisons across arrays of values. Executing Parabix2 on Core-i3 over Core2 results in an average performance improvement of 17% in bit stream processing whereas migrating Parabix2 from Core-i3 to SandyBridge results in a 22% average performance gain. Bit space measurements are stable and consistent across each of the source inputs and cores. Postprocessing operations demonstrate data dependent variance. Performance gains from 18% to 31% performance are observered in migrating Parabix2 from Core2 to Core-i3; 0% to 17% performance from Core-i3 to SandyBridge. For the purpose of comparison, Figure 9 (b) shows the performance of the Expat parser on each of the processor cores. A performance improvement of less than 5% is observed when executing Expat on Core-i3 over Core2 and less than 10% on SandyBridge over Core-i3.

Overall, Parabix2 scales better than Expat. Simply executing identical Parabix2 object code on SandyBridge results in an overall performance improvement up to 26%. Additional performance aspects of Parabix2 on SandyBridge with AVX instructions are discussed in the following sections.



(a) Avg. Power of Parabix on various hardware (Watts)



(b) Avg. Energy Consumption on various hardware (nJ per kB)

7.2 Power and Energy

Figure 10(a) shows the average power consumption of Parabix2 over each workload and as executed on each of the processor cores — Core2, Core-i3 and SandyBridge. Average power consumption on Core2 is 32 watts. Execution on Core-i3 results in 30% power saving over Core2. SandyBridge saves 25% of the power compared with Core-i3 and consumes only 15 watts.

In XML parsing we observe energy consumption is dependent on processing time. That is, a reduction in processing time results in a directly proportional reduction in energy consumption. With newer processor cores comes improvements in application performance. As a result, Parabix2 executed on SandyBridge consumes 72% to 75% less energy than Parabix2 on Core2.

8 Scaling Parabix2 for AVX

In this section, we discuss the scalability and performance advantages of our 256-bit AVX (Advanced Vector Extensions) Parabix2 port. Parabix2 originally targetted the 128-bit SSE2 SIMD technology available on all modern 64-bit Intel and AMD processors but has recently been ported to AVX. AVX technology is commercially available on the latest the SandyBridge microarchitecture Intel processors.

8.1 Three Operand Form

In addition to the widening of 128-bit operations to 256-bit operations, AVX technology uses a non-destructive 3-operand instruction format. Previous SSE implementations used a destructive 2-operand

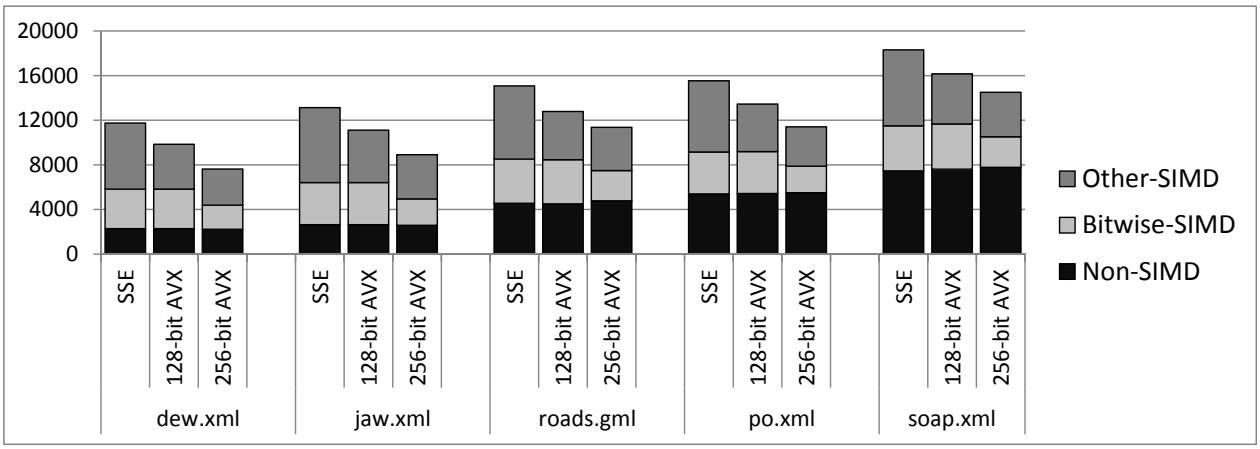


Figure 10: Parabix2 Instruction Counts (y-axis: Instructions per kB)

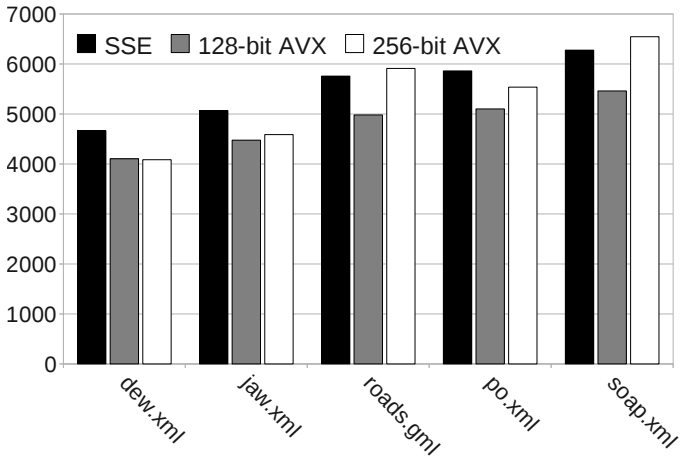


Figure 11: Parabix2 Performance (y-axis: CPU Cycles per kB)

instruction format. In the 2-operand format a single register is used as both a source and destination register. For example, $a = a$ `[op]` b. As such, 2-operand instructions that require the value of both a and b, must either copy an additional register value beforehand, or reconstitute or reload a register value afterwards to recover the value. With the 3-operand format, output may now be directed to the third register independently of the source operands. For example, $c = a$ `[op]` b. By avoiding the copying or reconstituting of operand values, a considerable reduction in instruction count in the form of reduced load and store instructions is possible. 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 AVX 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 Parabix2 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. Nevertheless, with loads, stores and bitwise logic comprising a major portion of the Parabix2 SIMD instruction mix, a substantial reduction in instruction count and consequent performance improvement was anticipated but not achieved.

8.3 Performance Results

We implemented two versions of Parabix2 using AVX technology. The first was simply the recompilation of the existing Parabix2 source code written to take advantage of the 3-operand form of AVX instructions while retaining a uniform 128-bit SIMD processing width. The second involved rewriting the core library functions of Parabix2 to leverage the 256-bit AVX operations wherever possible and to simulate the remaining operations using pairs of 128-bit operations.

Figure 10 shows the reduction in instruction counts achieved in these two versions. For each workload, the base instruction count of the Parabix2 binary compiled in SSE-only mode is shown with the caption "sse," the version obtained by simple recompilation with AVX-mode enabled is labeled "128-bit avx," and the version reimplemented to use 256-bit operations wherever possible is labelled "256-bit avx." The instruction counts are divided into three classes. The "non-SIMD" operations are the general purpose instructions that use neither SSE nor AVX technology. The "bitwise SIMD" class comprises the bitwise logic operations, that are available in both 128-bit form and 256-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 with 256-bit AVX technology.

Note that, in each workload, the number of non-SIMD instructions remains relatively constant with each workload. As may be expected, however, 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. Ideally one one may

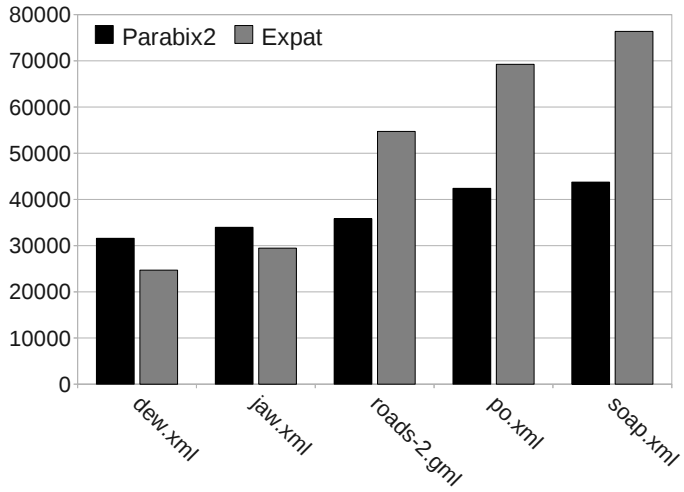
expect up to a 50% reduction in these instructions versus the 128-bit AVX. The actual reduction measured was 32%–39% depending on workload. Because some bitwise logic is needed in implementation of simulated 256-bit operations, the full 50% reduction in bitwise logic was not achieved.

The "other SIMD" class shows a substantial 30%-35% reduction with AVX 128-bit technology compared to SSE. This reduction is due to eliminated copies or reloads when SIMD operations are compiled using 3-operand AVX form versus 2-operand SSE form. A further 10%–20% reduction is observed with Parabix2 version rewritten to use 256-bit operations.

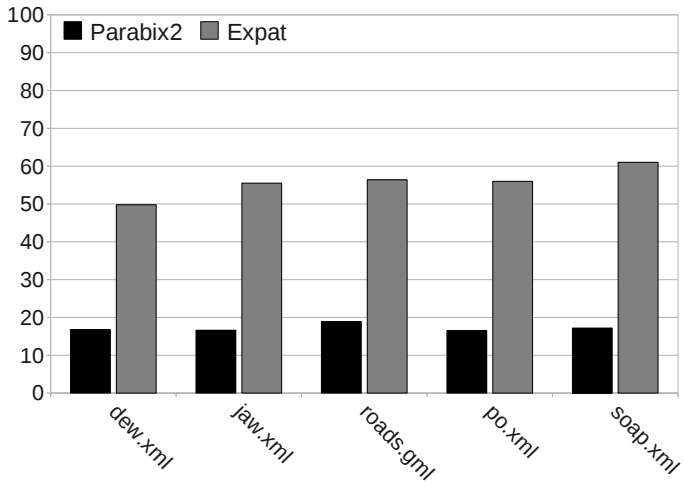
While the successive reductions in SIMD instruction counts are quite dramatic with the two AVX implementations of Parabix2, the performance benefits are another story. As shown in Figure 11, the benefits of the reduced SIMD instruction count are achieved only in the AVX 128-bit version. In this case, 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 bizzarely, the performance of Parabix2 in the 256-bit AVX implementation does not improve significantly and actually degrades for files with higher markup density (average 10%). Dewiki.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 implemention 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 SIMD throughput. If these latency issues can be addressed in future AVX implementations, further substantial performance and energy benefits could be realized in XML parsing with Parabix2.

9 Parabix on Mobile Platforms

The Samsung Galaxy Tab GT-P1000M device 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. In addition to the standard feature set found in such low-power 32-bit microprocessors, the S5PC110 includes the ARM NEON general-purpose SIMD engine. ARM NEON makes available a 128-bit SIMD instruction set similar in functionality to Intel SSE3 instruction set. In this section, we present our performance comparison of a NEON-based port of Parabix2 versus the Expat parser, and executed on



(a) ARM Neon Performance



(b) Performance ARM Neon vs Core i3 SSE.

the Samsung Galaxy Tab GT-P1000M hardware. Xerces is excluded from this portion of our study due to the complexity of the cross-platform build process in porting native C/C++ applications to the Android platform.

9.1 Performance Results

Migration of Parabix2 to the Android platform began with the retargetting of a subset of the Parabix2 IDISA SIMD library for ARM NEON. This library code was cross-compiled for Android using the Android NDK. The Android NDK is a companion tool to the Android SDK that allows developers to build performance-critical portions of applications in native code. The majority of the Parabix2 SIMD functionality ported directly. However, for a small subset of the SIMD functions of Parabix2 NEON equivalents did not exist. In such cases we simply simulated logical equivalencies using the available the instruction set.

A comparison of Figure 12(a) and Figure 9(a) demonstrates that the performance of both Parabix2 and Expat degrades substantially on Cortex-A8. This result was expected given the combarably performance limited Cortex-A8 hardware architecture. Surprisingly on Cortex-A8 Expat outperforms Parabix2 on each of the lower markup density workloads, dew.xml and jaw.xm. On the remaining higher-density workloads, Parabix2 performs only moderately better than Expat. The higher latency of the NEON instructions on Cortex-A8 is the likely contributor to this loss in performance. A more interesting aspect of this result is demonstrated in a comparison of Figure 12(b) and Figure 12(b). These figure demonstrate that the

relative performance of each parser degrades in a relatively constant manner. That is, compared to the Core-i3, on the GT-P1000M, Parabix2 and Expat operate at approximately 17.2% and 55.7% efficiency respectively. Figure 12(b) shows that the baseline cost of Parabix2 operations implemented using the NEON instruction set— and thereby the baseline cost of Parabix2—is substantially higher on the Cortex-A8 processor. Given that Parabix2 was not designed with the limitations of the Cortex-A8 in mind, in the future a careful analysis of the cost of each instruction provided in the ARMv7 ISA may allow us to better utilize the hardware resources provided. In particular, future performance enhancement to ARM NEON could result in substantial overall improvement in Parabix2 execution time.

10 Multi-threaded Parabix

The general problem of addressing performance through multicore parallelism is the increasing energy cost. As discussed in previous sections, Parabix, which applies SIMD-based techniques can not only achieves better performance but consumes less energy. Moreover, using mulitiple cores, we can further improve the performance of Parabix while keeping the energy consumption at the same level.

A typical approach to parallelizing software, data parallelism, requires nearly independent data, However, the nature of XML files makes them hard to partition nicely for data parallelism. Several approaches have been used to address this problem. A preparsing phase has been proposed to help partition the XML document [20]. The goal of this preparsing is to determine the tree structure of the XML document so that it can be used to guide the full parsing in the next phase. Another data parallel algorithm is called ParDOM [21]. It first builds partial DOM node tree structures for each data segments and then link them using preorder numbers that has been assigned to each start element to determine the ordering among siblings and a stack to manage the parent-child relationship between elements.

Data parallelism approaches introduce a lot of overheads to solve the data dependencies between segments. Therefore, instead of partitioning the data into segments and assigning different data segments to different cores, we propose a pipeline parallelism strategy that partitions the process into several stages and let each core work with one single stage.

The interface between stages is implemented using a circular array, where each entry consists of all ten data structures for one segment as listed in Table 3. Each thread keeps an index of the array (I_N), which is

		Data Structures									
		data_buffer	basis_bits	u8	lex	scope	ctCDPI	ref	tag	xml_names	err_streams
Stage1	read_data transposition classification	write read	 write read		 write						
Stage2	validate_u8 gen_scope parse_CtCDPI parse_ref		read	write	 read read read	 write read read	 write read	 write			 write
Stage3	parse_tag validate_name gen_check			 read read	read read read	read read	read read read	 read 	write read read	 write read	 write write
Stage4	postprocessing	read			read		read	read			read

Table 3: Relationship between Each Pass and Data Structures

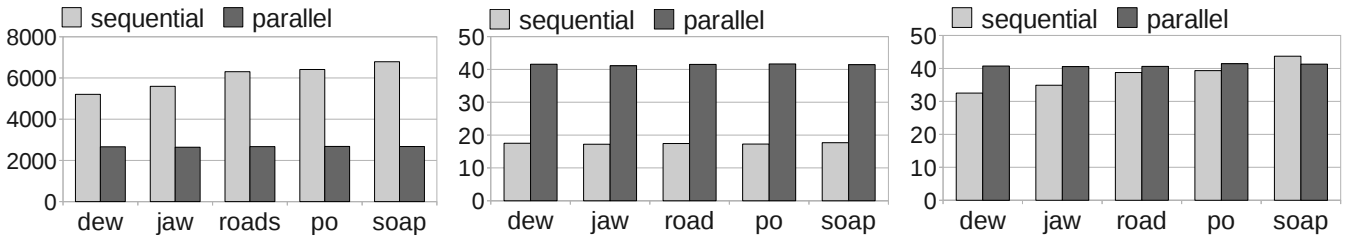
compared with the index (I_{N-1}) kept by its previous thread before processing the segment. If I_N is smaller than I_{N-1}, thread N can start processing segment I_N, otherwise the thread keeps reading I_{N-1} until I_{N-1} is larger than I_N. The time consumed by continuously loading the value of I_{N-1} and comparing it with I_N will be later referred as stall time. When a thread finishes processing the segment, it increases the index by one.

Figure 12 demonstrates the XML well-formedness checking performance of the multi-threaded Parabix in comparison with the single-threaded version. The multi-threaded Parabix is more than two times faster and runs at 2.7 cycles per input byte on the SandyBridge machine.

Figure 12(d) shows the average power consumed by the multi-threaded Parabix in comparison with the single-threaded version. By running four threads and using all the cores at the same time, the power consumption of the multi-threaded Parabix is much higher than the single-threaded version. However, the energy consumption is about the same, because the multi-threaded Parabix needs less processing time. In fact, as shown in Figure 12(e), parsing soap.xml using multi-threaded Parabix consumes less energy than using single-threaded Parabix.

11 Conclusion

This paper has examined energy efficiency and performance characteristics of four XML parsers considered over three generations of Intel processor architecture and shown that parsers based on parallel bit stream technology have dramatically better performance, energy efficiency and scalability than tradi-



(c) Performance (Cycles / kByte) (d) Avg. Power Consumption (e) Avg. Energy Consumption (nJ / Byte)

Figure 12: Multithreaded Parabix

tional byte-at-a-time parsers widely deployed in current software. Based on a novel application of the short vector SIMD technology commonly found in commodity processors of all kinds, parallel bit stream technology scales well with improvements in processor SIMD capabilities. With the recent introduction of the first generation of Intel processors that incorporate AVX technology, the change to 3-operand form SIMD operations has delivered a substantial benefit for the Parabix2 parsers simply through recompilation. Restructuring of Parabix2 to take advantage of the 256-bit SIMD capabilities also delivered a substantial reduction in instruction count, but without corresponding performance benefits in the first generation of AVX implementations.

There are many directions for further research. These include compiler and tools technology to automate the low-level programming tasks inherent in building parallel bit stream applications, widening the research by applying the techniques to other forms of text analysis and parsing, and further investigation of the interaction between parallel bit stream technology and processor architecture. Two promising avenues include investigation of GPGPU approaches to parallel bit stream technology and the leveraging of the intraregister parallelism inherent in this approach to also take advantage of the intrachip parallelism of multicore processors.

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

[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. 12

[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. 12

[4] W. Bircher and L. John. Complete system power estimation: A trickle-down approach based on performance events. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 158 –168, Apr. 2007. 12

[5] 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. 2

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

[7] R. Cameron, K. Herdy, and E. Amiri. Parallel bit stream technology as a foundation for XML parsing performance. In *International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*, Aug. 2009. 8

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

[9] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich. Parallel parsing with bitstream addition: An XML case study. Technical Report TR 2010-11, Simon Fraser University, School of Computing Sciece, October 2010. 6, 9

[10] 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. 6, 7, 8

[11] J. Clark. The Expat XML Parser. http://expat.sourceforge.net/. 6, 12

[12] F. Corporation. Fluke Clamp Meters. http://www.fluke.com/. 14

[13] B. DuCharme. Documents vs. data, schemas vs. schemas. In *XML 2004*, Washington D.C., 2004. 5

[14] 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. 2

[15] R. D. C. et. al. Parabix1. http://parabix.costar.sfu.ca/. 12

[16] R. D. C. et. al. Parabix2. http://parabix.costar.sfu.ca/. 12

[17] A. S. Foundation. Xerces C++ Parser. http://xerces.apache.org/xerces-c/. 6, 12

[18] 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. 2

[19] K. S. Herdy, D. S. Burggraf, and R. D. Cameron. High performance GML to SVG transformation for the visual presentation of geographic data in web-based mapping systems. In *Proceedings of SVG Open 2008*, August 2008. 8

[20] W. Lu, Y. Pan, , and K. Chiu. A parallel approach to xml parsing. *The 7th IEEE/ACM International Conference on Grid Computing*, 2006. 24

[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. 24

[22] 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. 2

[23] 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. 6