Energy Efficiency and Scalability of XML Parsing Using SIMD

ABSTRACT

XML is a data format designed for documents as well as the representation of data structures. The simplicity and generality of the rules make it widely used in web services and database systems. Traditional XML parsers have been built around the byte-at-a-time model, in which they process every character token in the file in a sequential fashion. Unfortunately, the byte-at-time sequential model is a performance barrier in demanding applications, and is also energy-inefficient, making poor use of the wide registers and other parallelism features in modern processors.

This paper assesses the energy and performance of a new approach to XML parsing based on parallel bit stream technology. This method first converts the character steams into sets of parallel bitstreams and then exploits SIMD operations prevalent on modern CPUs. The first generation Parabix1 parser then uses bit-scan instructions over these streams to make multibyte moves in an otherwise sequential approach. The second generation Parabix2 technology adds further parallelism by replacing much of the sequential bit scanning with a parallel scanning approach based on bit-stream addition. We evaluate Parabix1 and Parabix2 against two widely-used XML parsers, James Clark's Expat and Apache's Xerces on three generations of x86 machines, including the new Intel SandyBridge. We show that Parabix2's speedup is 2×–7× over Expat and Xerces. In stark contrast to the energy expenditures necessary to realize performance gains through multicore parallelism, we also show that our Parabix parsers deliver energy savings directly in proportion to performance gains. We also assess the scalability advantages of SIMD processor improvements the different Intel machine generations, culminating with an evaluation of the 256-bit AVX technology in SandyBridge vs. the now legacy 128-bit SSE technology.

1. Introduction

Extensible Markup Language (XML) is a core technology standard of the World Wide Web Consortium (W3C) that provides a common framework for encoding and communicating structured information of all kinds. In applications ranging from Office Open XML in Microsoft Office to NDFD XML of the NOAA National Weather Service, from KML in Google Earth to Castor XML in the Martian Rovers, from ebXML for e-commerce data interchange to RSS for news feeds from web sites everywhere, XML plays a ubiquitous role in providing a common framework for data interoperability world-wide and beyond. As XML 1.0 editor Tim Bray is quoted in the W3C celebration of XML at 10 years, "there is essentially no computer in the world, desk-top, hand-held, or back-room, that doesn't process XML sometimes."

With all this XML processing, a substantial literature has arisen addressing XML processing performance in general and the performance of XML parsers in particular. Nicola and John specifically identified XML parsing as a threat to database performance and outlined a number of potential directions for potential performance improvements [19]. The nature of XML APIs was found to have a significant affect on performance with event-based SAX (Simple API for XML) parsers avoiding the tree construction costs of the more flexible DOM (Document Object Model) parsers [20]. The

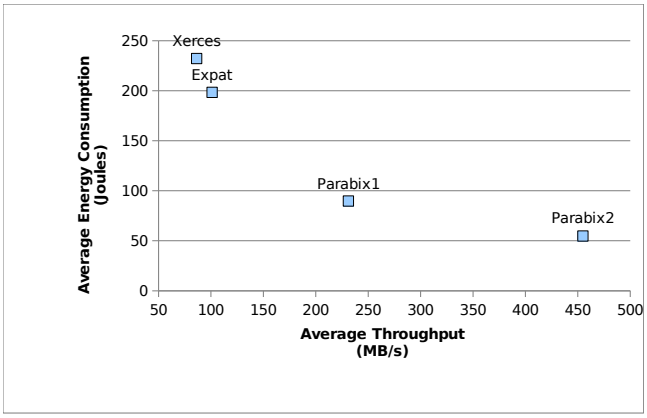


Figure 1: XML Parser Technology Energy vs. Performance

commercial importance of XML parsing spurred developments of hardware-based approaches including the development of a custom XML chip [17] as well as FPGA-based implementations [11]. As promising as these approaches may be for particular niche applications, however, it is still likely that the bulk of the world's XML processing workload will be carried out on commodity processors using software-based solutions.

To accelerate XML parsing performance in software, most recent work has focused on parallelization. The use of multicore parallelism for chip multiprocessors has attracted the attention of several groups [18, 21, 22], while SIMD (single-instruction multiple data) parallelism has been of interest to Intel in designing new SIMD instructions [16] as well as to the developers of parallel bit stream technology [5, 7, 8]. Each of these approaches has shown considerable performance benefits over traditional sequential parsing following the byte-at-a-time model.

With a focus on performance, however, relatively less attention has been paid to reducing energy consumption. For example, in addressing performance through multicore parallelism, one generally has to pay an energy price for performance gains because of the increased processing required for synchronization. A focus on reduction of energy consumption is a key topic in this paper, in which we study the energy and performance characteristics of several XML parsers across three generations of x86-64 processor technology. The parsers we consider are the widely used byte-at-a-time parsers Expat and Xerces as well the Parabix1 and Parabix2 parsers based on parallel bit stream technology. A compelling result is that the performance benefits of parallel bit stream technology translate directly and proportionately to substantial energy savings. Figure 1 is an energy-performance scatter plot showing the results we obtain for the four parsers.

The remainder of this paper is organized as follows. Section 2 presents background material on XML parsing and traditional parsing methods. Section 3 then reviews parallel bit stream technology as applied to XML parsing in the Parabix1 and Parabix2 parsers. Section 4 then introduces our methodology and approach for the performance and energy study tackled in the remainder of the paper. Section 5 presents a detailed performance evaluation on a Core-i3 processor as our primary evaluation platform, address-

ing a number of microarchitectural issues including cache misses, branch mispredictions, SIMD instruction counts and so on. Section 6 then looks at scalability and performance gains through three generations of Intel architecture culminating with performance assessment on our two week-old SandyBridge test machine. Section 7 looks specifically at issues in applying the new 256-bit AVX technology to parallel bit stream technology and notes that the major performance benefit seen so far is a result of the change to 3-operand instruction form. Section 8 concludes the paper with a discussion of ongoing work and further research directions.

2. Background

2.1 XML

In 1998, the W3C officially adopted XML as a standard. XML is a platform-independent data interchange format. 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 [4]. An example of XML document is as follows.

```
<?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

XML files can be classified as "document-oriented" or "data-oriented" [12]. 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" [4], 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.

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 are 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. Unfortunetly, textual data tends to consist of variable-length items sequenced in generally unpredictable patterns [7]; 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 [9]. Xerces-C was released in 1999 and is the foundation of the Apache XML project [14].

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 cummulative effect of branch mispredictions penalties are known to degrade XML parsing performance in proportion to the markup density of the source document [8] (i.e., the proportion of XML-markup vs. XML-data).

2.3 Parallel XML Parsing

In general, parallel XML acceleration methods comes 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 [22] and speculative p-DFAs [22]. 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, 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 into the SIMD classification. The Parabix parser 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" [7].

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.

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 packs, shifts, 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 [8].

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

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

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 character 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 varies between 64-bit for MMX, 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 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 bit scan operation. 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 [5, 8, 15].

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 cursor positions. Unlike the single-cursor approach of Parabix1 (and conceptually of all sequential XML parsers), Parabix2 processes multiple cursors in parallel. For example, using 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

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

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 [7].

source data	<t1>abc</t1><tag2/>
$N =$ Tag Names	.11.....11..1111..
$M_0 =$ [<]	1.....1.....
$M_1 = advance(M_0)$	.1.....1.....
$M_2 = scanto(M_1, N)$	...1.....1.
$M_3 = scanto(M_2, N)$	...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 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. 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 [1] (PMC) hardware events listed in the PMC Hardware Events subsection. Based on the findings of previous work [1–3] we have chosen

[1] 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.

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 [1] as well as more recent work in [2, 3] demonstrate that hardware-usage patterns have a significant impact on the energy consumption characteristics of an application [1–3]. 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.

4.1 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 [**?**]. 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 [13] 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) [14] is a validating open source XML parser written in C++ by the Apache project. Expat version 2.0.1 [9] is a non-validating XML parser library written in C.

4.2 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.

4.3 Platform Hardware

Intel Core2.

Intel Core2 processor, code name Conroe, produced by Intel. Table 2 gives the hardware description of the Intel Core2 machine.

Processor	Intel Core2 Duo processor 6400 (2.13GHz)
L1 Cache	32KB I-Cache, 32KB D-Cache
L2 Cache	2MB
Front Side Bus	1066 MHz
Memory	2GB
Hard disk	80GB SCSI
Max TDP	65W

Table 2: Core2

Intel Core-i3.

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 3 gives the hardware description of the Intel Core-i3 machine.

Processor	Intel i3-530 (2.93GHz)
L1 Cache	32KB I-Cache, 32K D-Cache
L2 Cache	256KB
L3 Cache	4-MB
Front Side Bus	1333 MHz
Memory	4GB
Hard disk	SCSI 1TB
Max TDP	73W

Table 3: Core-i3

Intel Core-i5.

Intel Core-i5 processor, code name SandyBridge produced by Intel. Table 4 gives the hardware description of the Intel Core-i3 machine.

Processor	Intel Sandybridge i5-2300 (2.80GHz)
L1 Cache	192 KB
L2 Cache	4 X 256KB
L3 Cache	6-MB
Front Side Bus	1333 MHz
Memory	6GB DDDR
Hard disk	SATA 1TB
Max TDP	95W

Table 4: SandyBridge

4.4 PMC Hardware Events

Each of the hardware events selected relates to performance and energy features associated with one or more hardware units. For example, total 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
- Cache Misses

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

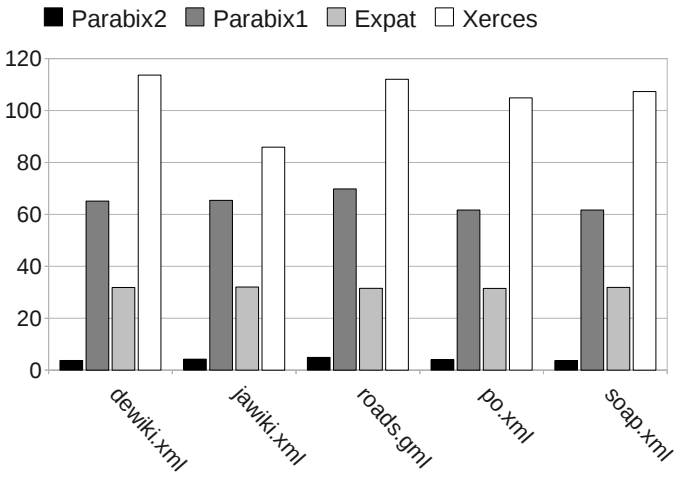


Figure 6: L1 Data Cache Misses on Core-i3 (y-axis: Cache Misses per kB)

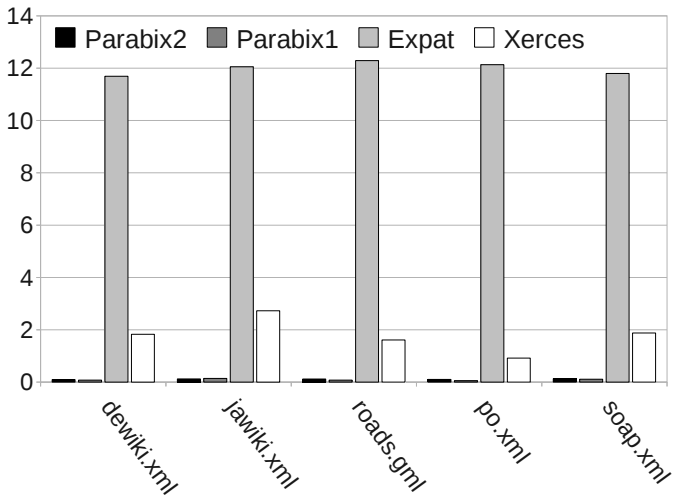


Figure 7: L2 Data Cache Misses on Core-i3 (y-axis: Cache Misses per kB)

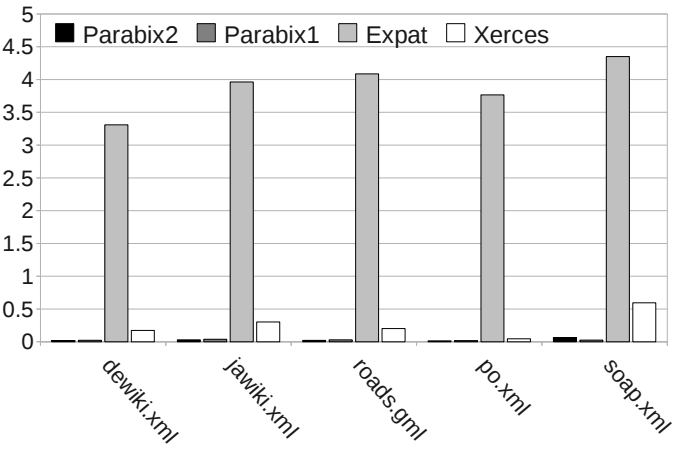


Figure 8: L3 Cache Misses on Core-i3 (y-axis: Cache Misses per kB)

4.5 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 [10].

5. Baseline Evaluation on Core-i3

5.1 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 6, Figure 7 and Figure 8 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.

5.2 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 generally cited as over 10 CPU cycles. As shown in Figure 10, the cost of branch mispredictions per XML byte for Expat can be over 7 cycles—this cost alone is equal to the total cost for Parabix2 to process each byte of XML given the same input.

But reducing the branch misprediction rate is difficult for text-based applications due to the variable-length nature of syntactic elements. Therefore, the goal is to reduce the total number of branches. However, traditional byte-at-a-time XML parsing requires a large number of inevitable branches. As shown in Figure 9, Xerces can have an average of 13 branches for each byte it processed on the high markup density file. Parabix1 minimizes the branches by using parallel bit streams for each 128-bit block but still requires a few branches for sequential scanning. Utilizing the new parallel scanning technique, Parabix2 is relatively branch-free, as shown in Figure 9. As a result, Parabix2 has minimal dependency on the markup density of the workloads.

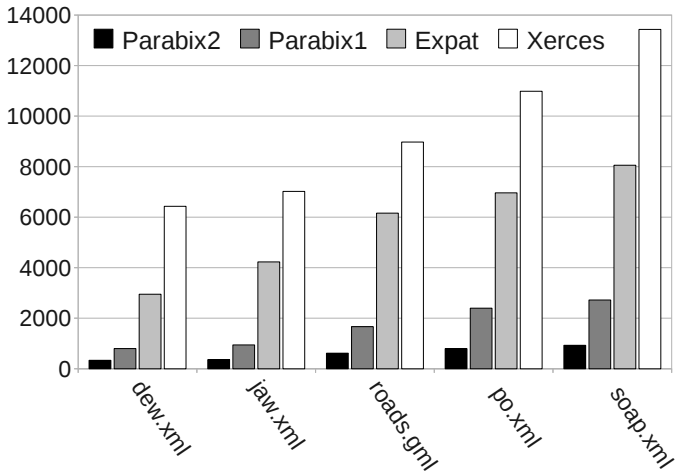


Figure 9: Branches on Core-i3 (y-axis: Branches per kB)

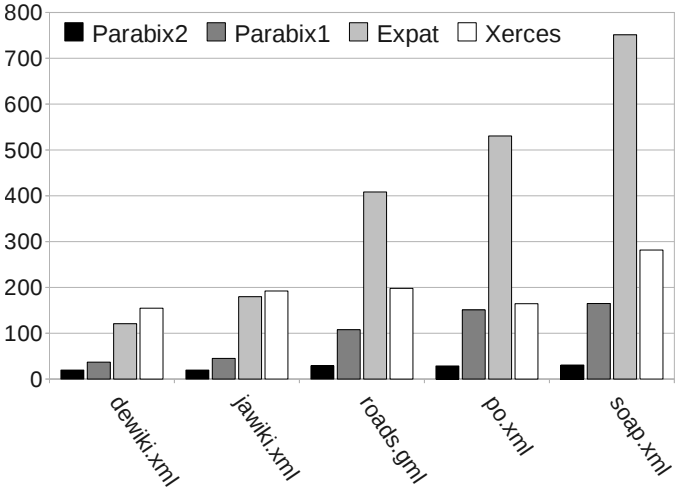


Figure 10: Branch Mispredictions on Core-i3 (y-axis: Branch Mispredictions per kB)

5.3 SIMD Instructions vs. Total Instructions

Parabix gains its performance by using parallel bitstreams, which are mostly generated and calculated by SIMD instructions. The ratio of executed SIMD instructions over total instructions indicates the amount of parallel processing we were able to achieve. Using Intel PIN, a dynamic binary instrumentation tool, we gathered the running instruction mix of each XML workload and classified the instructions as either vector (SIMD-based) instructions or non-vector (Non-SIMD-based) instructions. Figure 11 and Figure 12 shows the percentage of SIMD instructions of Parabix1 and Parabix2 . For Parabix1, 18% to 40% of the executed instructions consists of SIMD instructions. By using bistream addition for parallel scanning, Parabix2 uses 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.

5.4 CPU Cycles

Figure 13 shows the result of the overall performance evaluated as CPU cycles per thousand input bytes. Parabix1 is 1.5 to 2.5 times faster on document-oriented input and 2 to 3 times faster on data-oriented input compared with Expat and Xerces. Parabix2 is

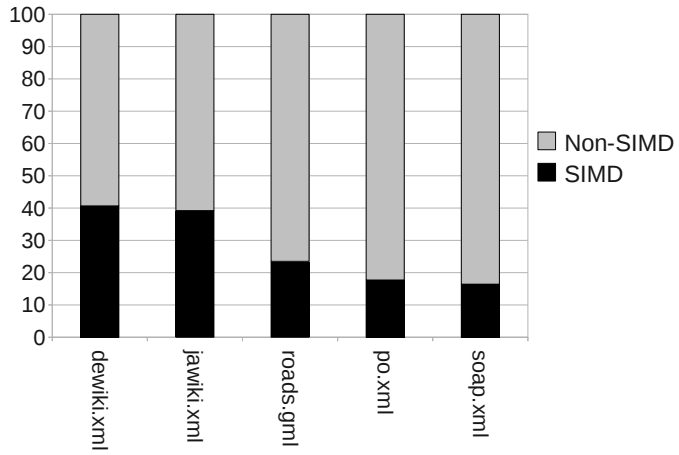


Figure 11: Parabix1 SIMD vs. Non-SIMD Instructions (y-axis: Percent SIMD Instructions

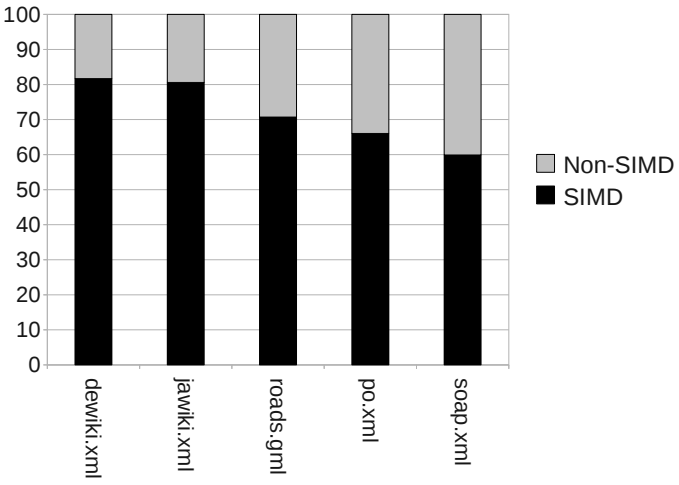


Figure 12: Parabix2 SIMD vs. Non-SIMD Instructions (y-axis: Percent SIMD Instructions)

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 down by higher markup density while Parabix with parallel processing is less affected. The comparison is not entirely fair for Xerces that transcodes input into UTF-16, which typically takes several cycles per byte. However, transcoding using parallel bitstreams can be much faster and it takes less than a cycle per byte to transcode ASCI3I files such as road.gml, po.xml and soap.xml [6].

5.5 Power and Energy

There is a growing concern of power consumption and energy efficiency. Chip producers not only work on improving the performance but also have worked hard to develop power efficient chips. We studied the power and energy consumption of Parabix in comparison with Expat and Xerces on Core-i3.

Figure 14 shows the average power consumed by the four different parsers. The average power of Core-i3 530 is about 21 watts. This model released by Intel last year has a good reputation for power efficiency. Parabix2 dominated by SIMD instructions uses only about 5% higher power than the other parsers.

The more interesting trend is energy, Figure 15 shows the energy consumption of the four different parsers. Although Parabix2

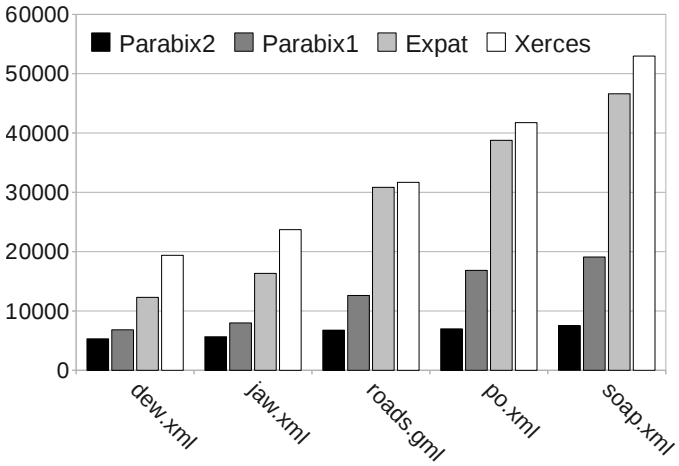


Figure 13: Processing Time on Core-i3 (y-axis: Total CPU Cycles per kB)

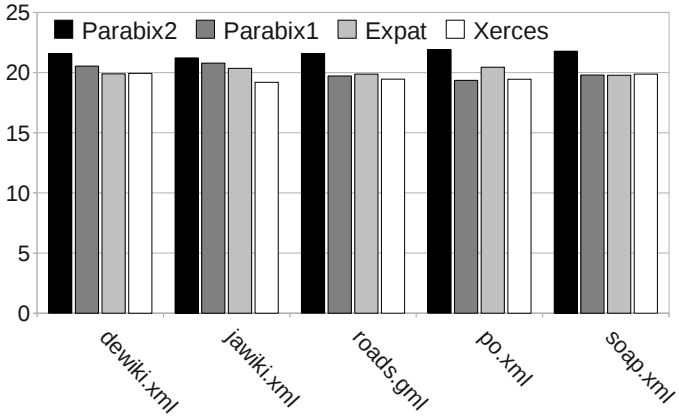


Figure 14: Average Power on Core-i3 (watts)

requires slightly more power (per instruction), its processing time is significantly lower and therefore consumes substantially less energy than the other parsers. Parabix2 consumes 50 to 75 nJ per byte while Expat and Xerces consumes 80nJ to 320nJ and 140nJ to 370nJ per byte seperately.

6. Scalability

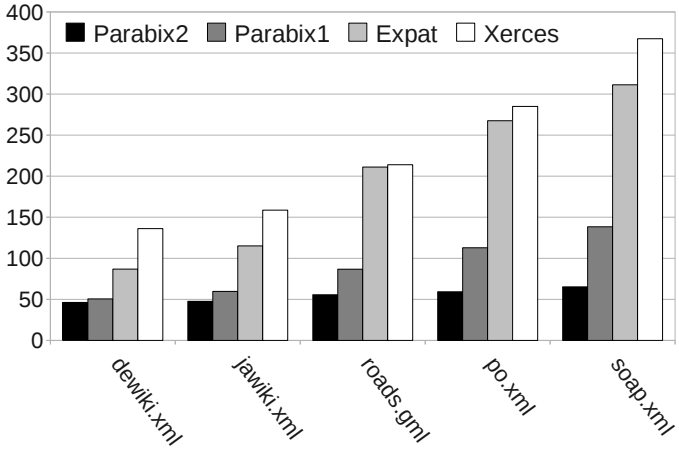
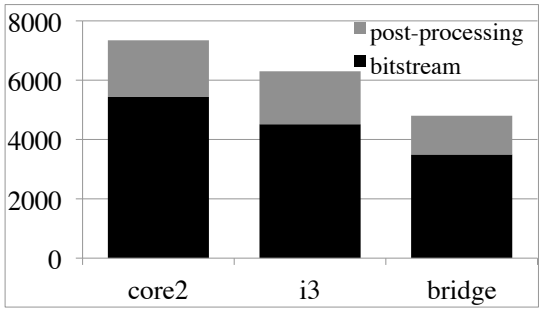
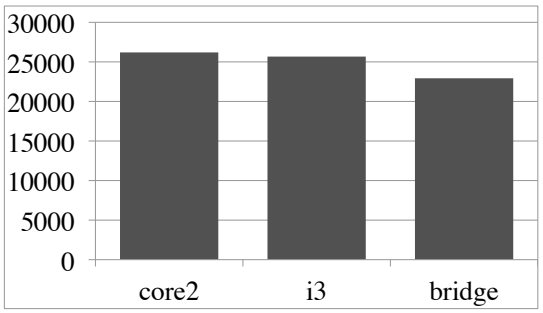


Figure 15: Energy Consumption on Core-i3 (μJ per kB)



(a) Parabix2



(b) Expat

Figure 16: Processing Time of Parabix and Expat (y-axis: Total CPU Cycles per KByte)

6.1 Performance

Figure 16 (a) shows the performance of Parabix2 on three different cores: Core2, Core-i3 and SandyBridge. The average processing time of the five workloads, which is evaluated as CPU cycles per thousand bytes, is divided up by bitstream parsing and byte space postprocessing. Bitstream parsing, which mainly consists of SIMD instructions, is able to achieve 17% performance improvement moving from Core2 to Core-i3; 22% performance improvement moving from Core-i3 to SandyBridge, which is relatively stable compared to postprocessing, which gains 18% to 31% performance moving from Core2 to Core-i3; 0 to 17% performance improvement moving from Core-i3 to SandyBridge.

As comparison, we also measured the performance of Expat on all the three cores, which is shown is Figure 16 (b). The performance improvement is less than 5% by running Expat on Core-i3 instead of Core2 and it is less than 10% by running on SandyBridge instead of Core-i3.

Parabix2 scales much better than Expat and is able to achieve an overall performance improvement up to 26% simply by running the same code on a newer core. Further improvement on SandyBridge with AVX will be discussed in the next section.

6.2 Power and Energy

The newer processors are not only designed to have better performance but also more energy-efficient. Figure 17 shows the average power when running Parabix2 on Core2, Core-i3 and SandyBridge with different input files. On Core2, the average power is about 32 watts. Core-i3 saves 30% of the power compared with Core2. SandyBridge saves 25% of the power compared with Core-i3 and consumes only 15 watts.

The energy consumption is further improved by better performance, which means a shorter processing time, as we moved to the newer cores. As a result, Parabix2 on SandyBridge cost 72% to 75% less energy than Parabix2 on Core2.

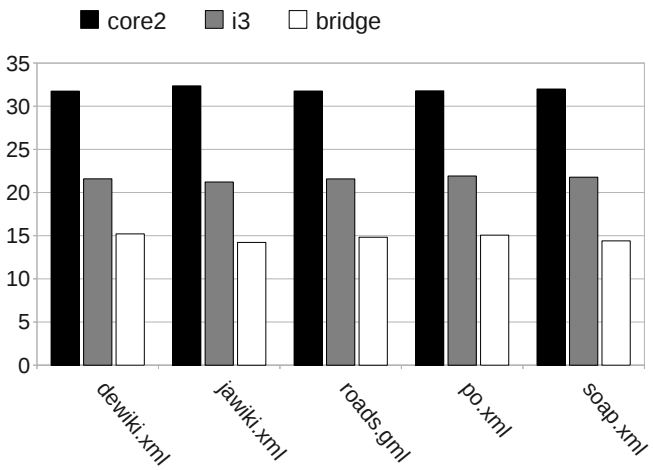


Figure 17: Average Power of Parabix2 (watts)

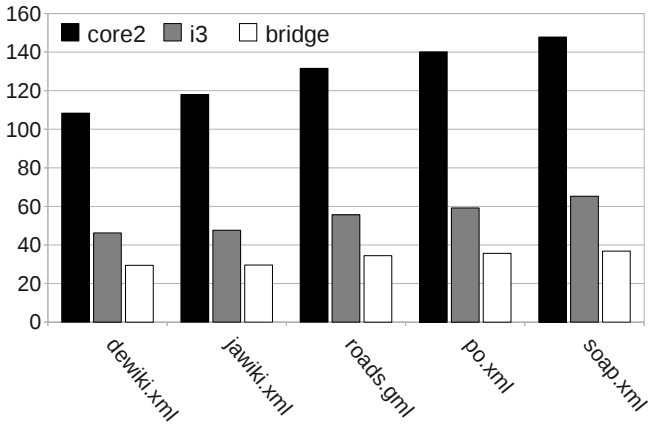


Figure 18: Energy consumption of Parabix2 (nJ/B)

7. Scaling Parabix2 for AVX Technology

Parabix2 was originally developed for 128-bit SSE2 technology widely and is available on all 64-bit Intel and AMD processors. In this section, we discuss the scalability and performance of Parabix2 to take advantage of the new 256-bit AVX (Advanced Vector Extensions) technology that has just become commercially available in the latest Intel processors based on the SandyBridge microarchitecture.

7.1 Three Operand Form

In addition to the introduction of 256-bit operations, AVX technology also makes a change in the structure of the base SSE instructions, moving from a destructive 2-operand form long used with SSE technologies to a nondestructive 3-operand form. In the 2-operand form, one register is used as both a source and destination register, equivalent to the assignment $a = a$ `[op]` b. Thus, whenever the subsequent instructions used the value of both a and b, one of them had to be copied beforehand, or reconstituted or reloaded afterwards in order to recover the value. With 3-operand form, output may be directed to a third register independent of the source operands, as reflected by the assignment $c = a$ `[op]` b. By avoiding the copying or reconstituting of operand values, a considerable reduction in instruction count may be possible. AVX technology makes available the 3-operand form both with the new 256-bit operations as well as with base 128-bit operations of SSE.

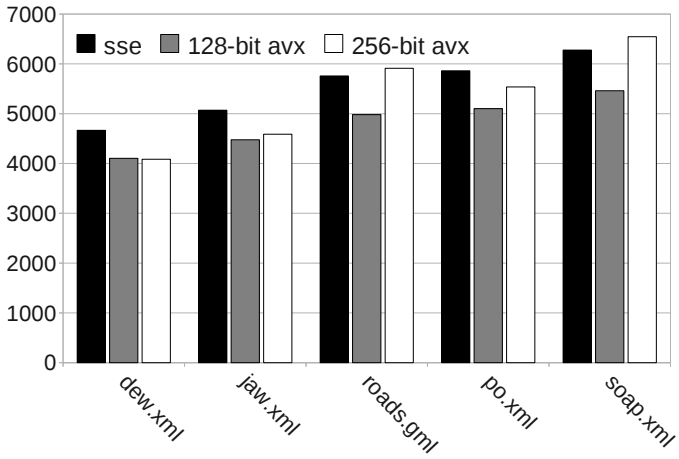


Figure 20: Parabix2 Performance (y-axis: CPU cycles per KB)

7.2 256-bit Operations

With the introduction of 256-bit SIMD registers with AVX technology, one might ideally expect up to a 50% reduction in the instruction count for the SIMD workload of Parabix2. However, in the SandyBridge implementation, Intel has focused on implementing floating point operations as opposed to the integer based operations. That is, 256-bit SIMD is available for loads, stores, bitwise logic and floating operations, while SIMD integer operations and shifts are only available in 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.

7.3 Performance Results

We implemented two versions of Parabix2 using AVX technology. The first was simply the recompilation of the existing Parabix2 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 core library functions for Parabix2 to use 256-bit AVX operations wherever possible and to simulate the remaining operations using pairs of 128-bit operations.

Figure 19 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 "avx 128-bit," and the version reimplemented to use 256-bit operations wherever possible is labelled "avx 256-bit." 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.

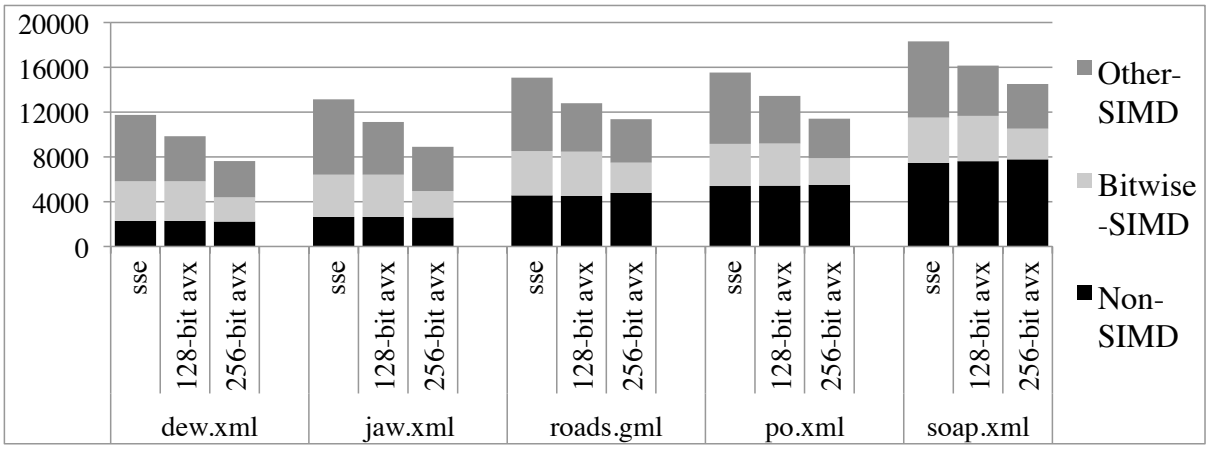


Figure 19: Parabix2 Instruction Counts (y-axis: Instructions per KByte)

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 20, 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.

8. 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 traditional 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.

9. References

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

[2] 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.

[3] 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.

[4] 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] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

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

[15] 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.

[16] Z. Lei. XML parsing accelerator with Intel streaming SIMD extensions 4 (Intel SSE4). http://software.intel.com/en-us/articles/xml-parsing-accelerator-with-intel-streaming-simd-extensions-4-intel-sse4/, 2008.

[17] 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.

[18] 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.

[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] Perkins, E. and Kostoulas, M. and Heifets, A. and Matsa, M. and Mendelsohn, N. Performance Analysis of XML APIs. In *XML 2005*, Atlanta, Georgia, Nov. 2005.

[21] B. Shah, P. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for XML DOM parsing. In Z. BellahsÃĺne, 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.

[22] 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.