

How to compute efficiently on commodity processors :

The Parabix XML parser Story.

Paper ID ****

Abstract

XML is a set of rules for the encoding of documents in machine-readable form. The simplicity and generality of the rules make it widely used in web services and database systems. Traditional XML parsers are built around a byte-at-a-time processing model where each character token of an XML document is examined in sequence. Unfortunately, the byte-at-a-time sequential model is a performance barrier in more demanding applications, is energy-inefficient, and makes poor use of the wide SIMD registers and other parallelism features of modern processors.

This paper assesses the energy and performance of a new approach to XML parsing, based on parallel bit stream technology, and as implemented on successive software generations of the Parabix XML parser. In Parabix, we first convert character streams into sets of parallel bit streams. We then exploit the SIMD operations prevalent on commodity-level hardware for performance. The first generation Parabix1 parser exploits the processor built-in *bitscan* instructions over these streams to make multibyte moves but follows 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, and across three generations of x86 machines, including the new Intel SandyBridge. We show that Parabix2's speedup is $2\times$ – $7\times$ 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 in direct proportion to the gains in performance. In addition, we assess the scalability advantages of SIMD processor improvements across Intel processor generations, culminating with an evaluation of the 256-bit AVX technology in SandyBridge versus the now legacy 128-bit SSE technology.

1 Introduction

Classical Dennard Scaling [] which ensured that voltage scaling would enable us to keep all of transistors afforded by Moore's law active, has currently stopped. 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 and processor cores in multiIntel multicores provide capability to boost core speeds 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 aiming to increase performance with larger number of cores. Unfortunately,

0057 given the levels of parallelism [4] in applications, that multicores can exploit it is not certain up to how
0058 many cores we can continue scaling our chips [13]. This is because exploiting parallelism across multiple
0059 cores tends to require heavyweight threads that are difficult to manage and synchronize.
0060
0061
0062

0063 The desire to improve the overall efficiency of computing is pushing designers to explore customized
0064 hardware [17, 19] that accelerate specific parts of an application while reducing the overheads present
0065 in general-purpose processors. They seek to exploit the transistor bounty to provision many different
0066 accelerators and keep only the accelerators needed for an application active while switching-off others on
0067 the chip to save power consumption. While promising, given the fast evolution of languages and software,
0068 its hard to define a set of fixed-function hardware for commodity processors. Furthermore, the toolchain to
0069 create such customized hardware is itself a hard research challenge. We believe that software, applications,
0070 and runtime models themselves can be refactored to significantly improve the overall computing efficiency
0071 of commodity processors.
0072
0073
0074
0075
0076
0077
0078
0079
0080

0081 In this paper, we demonstrate with an XML parser that changes to the underlying algorithm and com-
0082 pute model can significantly improve the efficiency on commodity processors. We achieve this efficiency
0083 by carefully redesigning the algorithm to exploit Parallel Bitstream runtime framework (Parabix) that
0084 exploits the SIMD extensions (SSE/AVX on x86, Neon on ARM) on commodity processors. The Para-
0085 bix framework exploits modern instructions in the processor ISA that can execute 10s of operations (on
0086 multiple character streams) in a single instruction and amortizes the overhead of general-purpose pro-
0087 cessor. Parabix also minimizes or eliminate branches entirely resulting in a more efficient pipeline and
0088 and improves overall register/cache utilization which minimizes energy wasted on data transfers. Parabix
0089 SSE/AVX exploits also include sophisticated instructions that enable the algorithm to pack and unpack
0090 the data elements from the registers which makes the overall cache access behavior of the application
0091 regular resulting in significantly fewer misses and better utilization. Overall as summarized by Figure 1
0092 our Parabix-based XML parser improves the performance by $?\times$ and energy efficiency by $?\times$ compared
0093 to widely-used software parsers and approaching the performance of $?cycles/input - byte$ performance of
0094 ASIC XML parsers [].¹
0095
0096
0097
0098
0099
0100
0101
0102
0103
0104
0105
0106
0107
0108

0109 XML is a particularly interesting application; it is a standard of the web consortium that provides
0110 a common framework for encoding and communicating data. XML provides critical data storage for
0111
0112

0113 ¹The actual energy consumption of the XML ASIC chips is not published by the companies.

0114 applications ranging from Office Open XML in Microsoft Office to NDFD XML of the NOAA National
0115 Weather Service, from KML in Google Earth to Castor XML in the Martian Rovers, a XML data in
0116 Android phones. XML parsing efficiency is important for multiple application areas; in server workloads
0117 the key focus is on overall transactions per second while in applications in the network switches and cell
0118 phones latency and the energy cost of parsing is of paramount importance. Software-based XML parsers
0119 are particularly inefficient and consist of giant *switch-case* statements, which waste processor resources
0120 processor since they introduce input-data dependent branches. They also have poor cache efficiency since
0121 they sift forward and backward through the input-data stream trying to match the parsed tags. XML
0122 ASIC chips have been around for over 6 years, but typically lag behind CPUs in technology due to cost
0123 constraints. Our focus is how much can we improve performance of the XML parser on commodity
0124 processors with Parabix technology.
0125

0126 Overall we make the following contributions in this paper.
0127

0128 1) We develop an XML parser that demonstrates the impact of redesigning the core of an application
0129 to make more efficient use of commodity processors. We compare the Parabix-XML parser against con-
0130 ventional parsers and demonstrate the improvement in overall performance and energy efficiency. We
0131 also parallelize the Parabix-XML parser to enable the different stages in the parser to exploit SIMD
0132 units across all the cores. This further improves performance while maintaining the energy consumption
0133 constant with the sequential version.
0134

0135 2) We are the first to compare and contrast SSE/AVX extensions across multiple generation of Intel
0136 processors and show that there are performance challenges when using newer generation SIMD extensions,
0137 possibly due to their memory interface. We compare ARM's Neon against x86's SIMD extensions and
0138 comment on the latency of SIMD operations across these architectures.
0139

0140 3) Finally, we introduce a runtime framework, *Parabix*, that abstracts the SIMD specifics of the machine
0141 (e.g., register widths) and provides a language framework to enable applications to run efficiently on
0142 commodity processors. Parabix enables the general-purpose multicores to be used efficiently by an entirely
0143 new class of applications, text processing and parsing.
0144

0145 The remainder of this paper is organized as follows. Section ?? presents background material on XML
0146 parsing and provides insight into the inefficiency of traditional parsers on mainstream processors. Sec-
0147 tion ?? reviews parallel bit stream technology a framework to exploit sophisticated data parallel SIMD
0148
0149
0150
0151
0152
0153
0154
0155
0156
0157
0158
0159
0160
0161
0162
0163
0164
0165
0166
0167
0168
0169
0170

0171
0172
0173
0174
0175
0176
0177
0178
0179
0180
0181
0182
0183
0184
0185
0186
0187
0188
0189
0190
0191
0192
0193
0194
0195
0196
0197
0198
0199
0200
0201
0202
0203
0204
0205
0206
0207
0208
0209
0210
0211
0212
0213
0214
0215
0216
0217
0218
0219
0220
0221
0222
0223
0224
0225
0226
0227

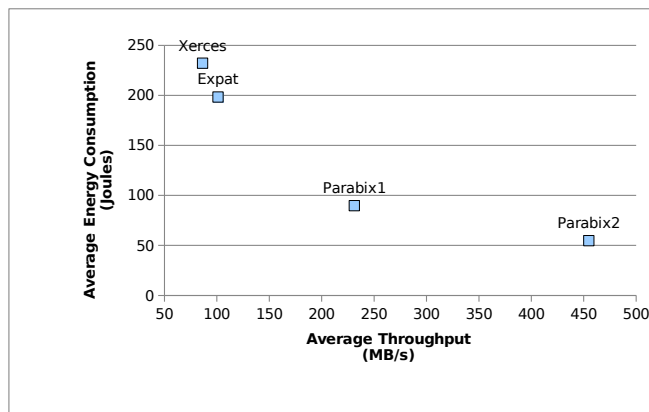


Figure 1: XML Parser Technology Energy vs. Performance

extensions on modern processors. Section 5 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 6 examines scalability and performance gains through three generations of Intel architecture culminating with a performance assessment on our two week-old SandyBridge test machine. We look 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 results from the change to the non-destructive three-operand instruction format.

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 [5].

XML files can be classified as “document-oriented” or “data-oriented” [12]. Document-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” [5], the latter often requires the use of an

0228 XML parser to extract the information within. The role of an XML parser is to transform the text-based
0229
0230 XML data into application ready data.
0231

```
0232 <?xml version="1.0"?>  
0233 <Products>  
0234   <Product ID="0001">  
0235     <ProductName Language="English">Widget</ProductName>  
0236     <ProductName Language="French">Bitoniau</ProductName>  
0237     <Company>ABC</Company>  
0238     <Price>$19.95</Price>  
0239   </Product>  
</Products>
```

0240
0241 Figure 2: Example XML Document
0242
0243

0244 2.2 Traditional XML Parsers 0245

0246 Traditional XML parsers process XML sequentially a single byte-at-a-time. Following this approach,
0247 an XML parser processes a source document serially, from the first to the last byte of the source file. Each
0248 character of the source text is examined in turn to distinguish between the XML-specific markup, such
0249 as an opening angle bracket ‘`<`’, and the content held within the document. The current character that
0250 the parser is processing is commonly referred to using the concept of a current cursor position. As the
0251 parser moves the cursor through the source document, the parser alternates between markup scanning, and
0252 data validation and processing operations. At each processing step, the parser scans the source document
0253 and either locates the expected markup, or reports an error condition and terminates. In other words,
0254 traditional XML parsers operate as complex finite-state machines that use byte comparisons to transition
0255 between data and metadata states. Each state transition indicates the context in which to interpret the
0256 subsequent characters. Unfortunately, textual data tends to consist of variable-length items sequenced in
0257 generally unpredictable patterns [8]; thus any character could be a state transition until deemed otherwise.
0258
0259

0260 Expat and Xerces-C are popular byte-a-time sequential parsers. Both are C/C++ based and open-source.
0261 Expat was originally released in 1998; it is currently used in Mozilla Firefox and provides the core func-
0262 tionality of many additional XML processing tools [10]. Xerces-C was released in 1999 and is the foun-
0263 dation of the Apache XML project [16].
0264
0265

0266 A major disadvantage of the sequential byte-at-a-time approach to XML parsing is that each XML
0267 character incurs at least one conditional branch. The cumulative effect of branch mispredictions penal-
0268 ties are known to degrade XML parsing performance in proportion to the markup density of the source
0269 document [9] (i.e., the proportion of XML-markup to XML-data).
0270
0271
0272
0273
0274
0275
0276
0277
0278
0279
0280
0281
0282
0283
0284

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 [20] and speculative p-DFAs [20]. 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” [8].

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 parses 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 position 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. $B_0 \dots B_7$ are the individual bit streams. The 0 bits in the bit streams are represented by periods as to emphasize the 1 bits.

```

0342
0343
0344         source data  <t1>abc</t1><tag2/>
0345         B0           ..1.1.1.1.1...11.1.
0346         B1           ...1.11.1..1...1111
0347         B2           11.1...111.111.1.11
0348         B3           1..1...11..11....11
0349         B4           1111...1.11111..1.1
0350         B5           11111111111111111111
0351         B6           .1..111..1...111...
0352         B7           .....
0353
0354

```

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 registers 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 [9].

Throughout the XML parsing process we must identify key XML characters. For example, the opening angle bracket character ‘*i*’. 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 ‘*i*’ 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 ‘*i*’ and end with either “*/i*” or “*i*” (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 proceed as follows. The first bit stream M_0 is created and the parser begins scanning the source data for an open angle bracket ‘*i*’, starting at position 1. Since the source data begins with ‘*i*’, 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

0399 cursor position at 2. The following *bitscan* operation takes the cursor position from M_1 and sequentially
 0400 scans every position until it locates either an ‘ ζ ’. It finds a ‘ ζ ’ at position 4 and returns that as the new
 0401 cursor position for M_2 . Calculating M_3 advances the cursor again, and the *bitscan* used to create M_4
 0402 locates the new opening angle bracket. This process continues in sequence until until all start tags are
 0403 validated. Unlike traditional parsers, these sequential operations are accelerated significantly since the
 0404 *bitscan* operation can skip up to w positions, where w is the processor word width in bits. This approach has
 0405 recently been applied to Unicode transcoding and XML parsing to good effect, with research prototypes
 0406 showing substantial speed-ups over even the best of byte-at-a-time alternatives [6, 9, 18].
 0407
 0408
 0409
 0410
 0411
 0412
 0413
 0414
 0415

0417	source data	<t1>abc</t1><tag2/>
0418	$M_0 = 1$	1.....
0419	$M_1 = \text{advance}(M_0)$.1.....
0420	$M_2 = \text{bitscan}(>')$...1.....
0421	$M_3 = \text{advance}(M_2)$1.....
0422	$M_4 = \text{bitscan}(<')$1.....
0423	$M_5 = \text{advance}(M_4)$1.....
0424	$M_6 = \text{advance}(M_5)$1.....
0425	$M_7 = \text{bitscan}(<')$1.....
0426	$M_8 = \text{advance}(M_7)$1.....
0427	$M_9 = \text{bitscan}(/')$1.
0428	$M_{10} = \text{advance}(M_9)$1

0431
 0432 Figure 4: Parabix1 Start Tag Validation
 0433
 0434
 0435

0436 3.2 Parabix2

0437
 0438 In Parabix2, the sequential single-cursor parsing approach using *bitscan* instructions is replaced by a
 0439 parallel parsing approach, that uses multiple cursors when possible, and bit stream addition operations to
 0440 advance multiple cursor positions in parallel. Unlike the single-cursor approach of Parabix1 (and concep-
 0441 tually of all sequential XML parsers), Parabix2 processes multiple cursors in parallel. For example, using
 0442 the source data from Figure 4, Figure 5 conceptually demonstrates the manner in which Parabix2 identi-
 0443 fies and advances each of the start tag bit streams. Unlike Parabix1, Parabix2 begins scanning by creating
 0444 two character-class bit streams, N , denoting the position of every alpha numeric character within the basis
 0445 stream, and M_0 , marking the position of every potential start tag in the bit stream. M_0 is advanced to create
 0446 M_1 , which is fed into the first *scanto* operation along with N . To handle variable length tag names, the
 0447
 0448
 0449
 0450
 0451
 0452
 0453
 0454
 0455

0456 *scanto* operation effectively locates the cursor positions of the end tags in parallel by adding M_1 to N , and
 0457 uses the bitwise AND operation of the negation of N to find only the true end tags of M_1 . Because an end
 0458 tag may end on an ‘/’ or ‘;’, *scanto* is called again to advance any cursor from ‘/’ to ‘;’. For additional
 0460 details, refer to the technical report [8].
 0461
 0462
 0463

```

0464
0465 source data          <t1>abc</t1><tag2/>
0466 N = Tag Names      .11.....11..1111..
0467 M0 = [<]          1.....1.....
0468 M1 = advance(M0) .1.....1.....
0469 M2 = scanthru(M1,A) ...1.....1..
0470
0471

```

0472 Figure 5: Parabix2 Start Tag Validation
 0473
 0474

0475
 0476 In general, the set of bit positions in a bit stream may be considered to be the current parsing positions
 0477 of multiple parses taking place in parallel throughout the source data stream. Although it is not explicitly
 0478 shown in these prior examples, error bit streams can be used to identify any well-formedness errors found
 0479 during the parsing process. Error positions are gathered and processed in as a final post processing step.
 0480 A further aspect of the parallel cursor method with bit stream addition is that the conditional branch
 0481 statements used to identify syntax error at each each parsing position are eliminated. Hence, Parabix2
 0482 offers additional parallelism over Parabix1 in the form of multiple cursor parsing and further reduces
 0483 branch misprediction penalties.
 0484
 0485
 0486
 0487
 0488
 0489
 0490
 0491

0492 3.3 Parallel Bit Stream Compilation

0493
 0494 While the description of parallel bit stream parsing in the previous section works conceptually on un-
 0495 bounded bit streams, in practice, a corresponding C implementation to process input streams into blocks
 0496 of size equal to the SIMD register width of the target processor is required. In our work, we leverage the
 0497 unbounded integer type of the Python programming language. Using a restricted subset of Python, we
 0498 prototype and validate the functionality of applications, such as XML validation and UTF-8 to UTF-16
 0499 transcoding. We then compile this Python code into equivalent block-at-a-time C code. The key ques-
 0500 tion becomes how to transfer information from one block to the next whenever token scans cross block
 0501 boundaries.
 0502
 0503
 0504
 0505
 0506
 0507
 0508
 0509

0510 The answer lies in carry bit propagation. Since the parallel *scanto* operation relies solely on bit-wise
 0511 addition and logical operations, block-to-block information transfer is captured in entirety by the carry bit
 0512

0513 associated with each underlying addition operation. Logical operations do not require information flow
0514 across block boundaries. Properly determining, initializing and inserting carry bits into a block-by-block
0515 implementation is tedious and error prone. Thus we have developed compiler technology to automatically
0516 transform parallel bit stream Python code to block-at-a-time C implementations. Details are beyond the
0517 scope of this paper, but are described in the on-line source code repository at parabix.costar.sfu.ca.
0518
0519
0520
0521
0522

0523 4 Methodology

0524 In this section we describe our methodology for the measurements and investigation of XML parser
0525 energy consumption and performance. In brief, for each of the four XML parsers under study we propose
0526 to measure and evaluate the energy consumption required to carry out XML well-formedness checking,
0527 under a variety of workloads, and as executed on three different Intel processors.
0528
0529
0530
0531
0532
0533
0534

0535 To begin our study we propose to first investigate each of the XML parsers in terms of the Performance
0536 Monitoring Counter² (PMC) hardware events listed in the PMC Hardware Events subsection. Based on
0537 the findings of previous work [1–3] we have chosen several key hardware performance events for which
0538 the authors indicate a strong correlation with energy consumption. In addition, we measure the runtime
0539 counts of SIMD instructions and bitwise operations using the Intel Pin binary instrumentation framework.
0540 Based on these data we gain further insight into XML parser execution characteristics and compare and
0541 contrast each of the Parabix parser versions against the performance of standard industry parsers.
0542
0543
0544
0545
0546
0547
0548

0549 The foundational work by Bellosa in [1] as well as more recent work in [2,3] demonstrate that hardware-
0550 usage patterns have a significant impact on the energy consumption characteristics of an application [1–3].
0551 Further, the authors demonstrate a strong correlation between specific PMC events and energy usage.
0552 However, each author differs slightly in their opinion of the exact set of PMCs to use.
0553
0554
0555
0556

0557 The following subsections describe the XML parsers under study, XML workloads, the hardware archi-
0558 tectures, PMC hardware events selected for measurement, and the energy measurement instrumentation
0559 set up. We analyze the performance of each of the XML parsers under study based on PMC hardware
0560 event counts and contrast their energy consumption measurements based on direct measurements.
0561
0562
0563
0564

0565 ²Performance Monitoring Counters are special-purpose registers available with most modern microprocessors. PMCs store
0566 the running count of specific hardware events, such as retired instructions, cache misses, branch mispredictions, and arithmetic-
0567 logic unit operations. PMCs can be used to capture information about any program at run-time and under any workload at a
0568 fine granularity.
0569

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

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 [14]. 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 [15] 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) [16] is a validating open source XML parser written in C++ by the Apache project. Expat version 2.0.1 [10] 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.

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

Processor	Intel Sandybridge i5-2300 (2.80GHz)
L1 Cache	32KB I-Cache, 32K D-Cache
L2 Cache	4 X 256KB
L3 Cache	6-MB
Front Side Bus	1333 MHz
Memory	6GB DDR
Hard disk	SATA 1TB
Max TDP	95W

Table 4: SandyBridge

- Branch Instructions
- Branch Mispredictions
- Integer Instructions
- SIMD Instructions
- Cache Misses

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 [11].

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

0741
0742
0743
0744
0745
0746
0747
0748
0749
0750
0751
0752
0753
0754
0755
0756
0757
0758
0759
0760
0761
0762
0763
0764
0765
0766
0767
0768
0769
0770
0771
0772
0773
0774
0775
0776
0777
0778
0779
0780
0781
0782
0783
0784
0785
0786
0787
0788
0789
0790
0791
0792
0793
0794
0795
0796
0797

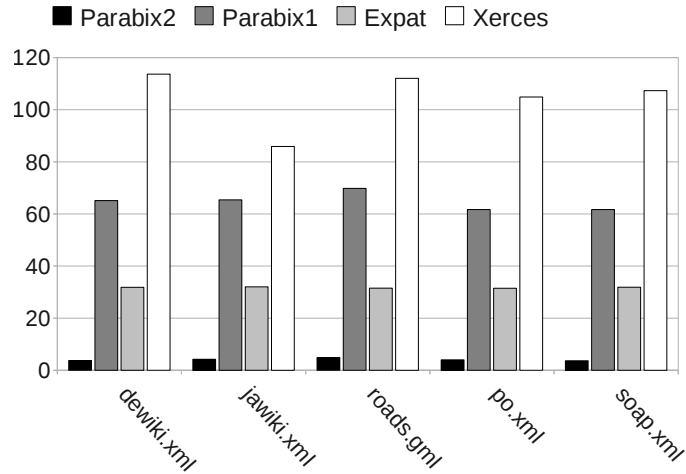


Figure 6: Core-i3 — L1 Data Cache Misses (y-axis: Cache Misses per kB)

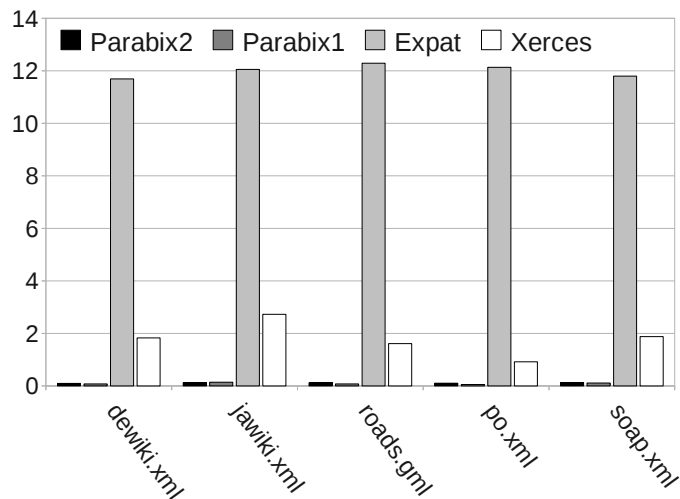


Figure 7: Core-i3 — L2 Data Cache Misses (y-axis: Cache Misses per kB)

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 commonly cited as over 10 CPU cycles. As shown in Figure 10, 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.

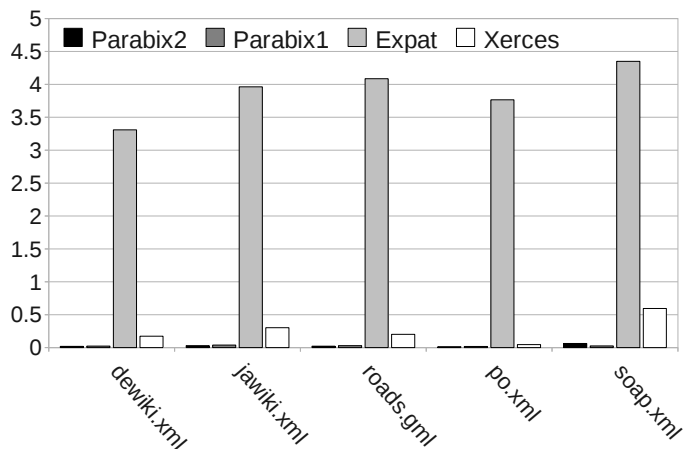


Figure 8: Core-i3 — L3 Cache Misses (y-axis: Cache Misses per kB)

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 characteristic, 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 9, 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 9, 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 9 displays this lack of data dependence via the constant number of branch mispredictions shown for each of the source XML files.

5.3 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

0855
 0856
 0857
 0858
 0859
 0860
 0861
 0862
 0863
 0864
 0865
 0866
 0867
 0868
 0869
 0870
 0871
 0872
 0873
 0874
 0875
 0876
 0877
 0878
 0879
 0880
 0881
 0882
 0883
 0884
 0885
 0886
 0887
 0888
 0889
 0890
 0891
 0892
 0893
 0894
 0895
 0896
 0897
 0898
 0899
 0900
 0901
 0902
 0903
 0904
 0905
 0906
 0907
 0908
 0909
 0910
 0911

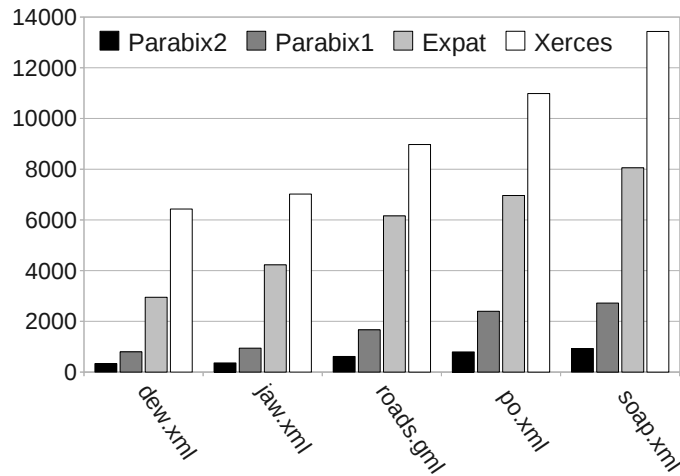


Figure 9: Core-i3 — Branch Instructions (y-axis: Branches per kB)

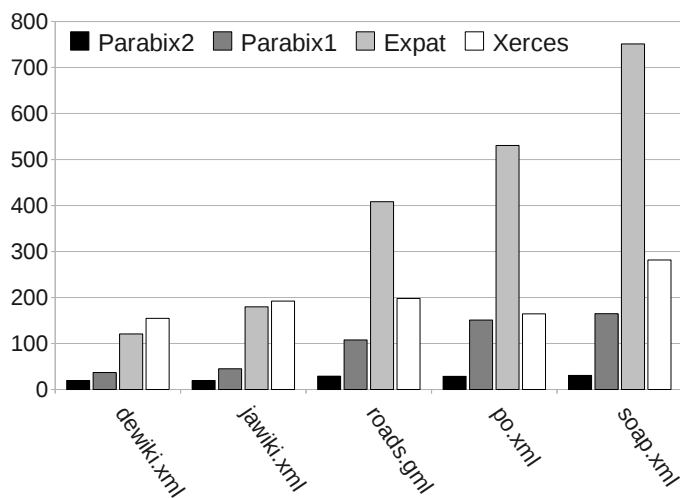


Figure 10: Core-i3 — Branch Mispredictions (y-axis: Branch Mispredictions per kB)

degree to which Parabix achieves parallelism over the byte-at-a-time approach.

Using the Intel Pin tool, we gather the dynamic instruction mix for each XML workload, and classify instructions as either vector (SIMD) or non-vector instructions. Figures 11 and 12 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.

0912
 0913
 0914
 0915
 0916
 0917
 0918
 0919
 0920
 0921
 0922
 0923
 0924
 0925
 0926
 0927
 0928
 0929
 0930
 0931
 0932
 0933
 0934
 0935
 0936
 0937
 0938
 0939
 0940
 0941
 0942
 0943
 0944
 0945
 0946
 0947
 0948
 0949
 0950
 0951
 0952
 0953
 0954
 0955
 0956
 0957
 0958
 0959
 0960
 0961
 0962
 0963
 0964
 0965
 0966
 0967
 0968

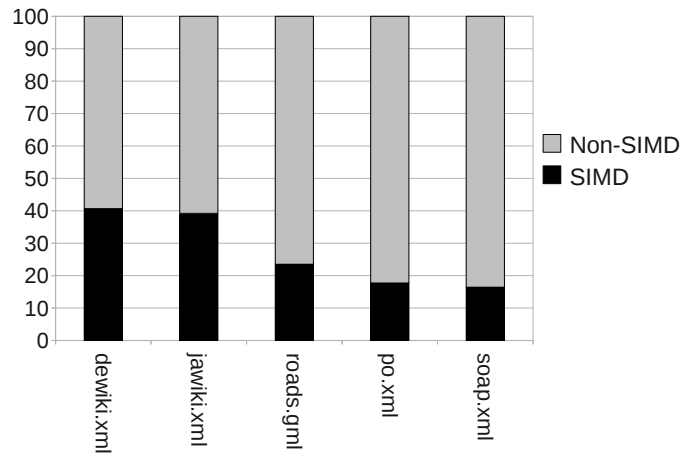


Figure 11: Parabix1 — SIMD vs. Non-SIMD Instructions (y-axis: Percent SIMD Instructions)

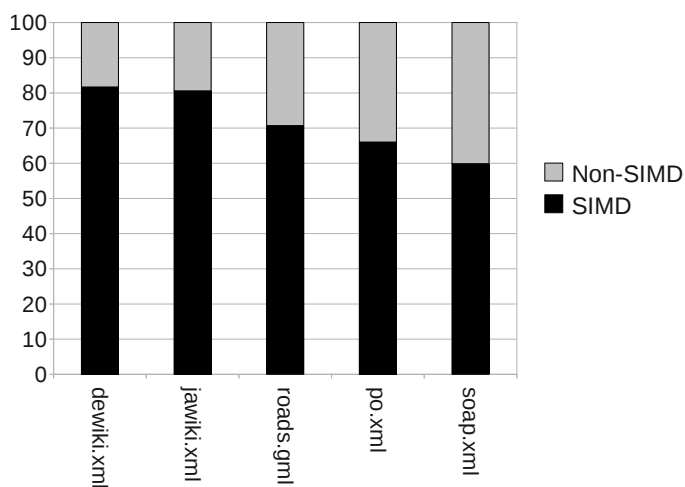


Figure 12: Parabix2 — SIMD vs. Non-SIMD Instructions (y-axis: Percent SIMD Instructions)

5.4 CPU Cycles

Figure 13 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. [7].

0969
 0970
 0971
 0972
 0973
 0974
 0975
 0976
 0977
 0978
 0979
 0980
 0981
 0982
 0983
 0984
 0985
 0986
 0987
 0988
 0989
 0990
 0991
 0992
 0993
 0994
 0995
 0996
 0997
 0998
 0999
 1000
 1001
 1002
 1003
 1004
 1005
 1006
 1007
 1008
 1009
 1010
 1011
 1012
 1013
 1014
 1015
 1016
 1017
 1018
 1019
 1020
 1021
 1022
 1023
 1024
 1025

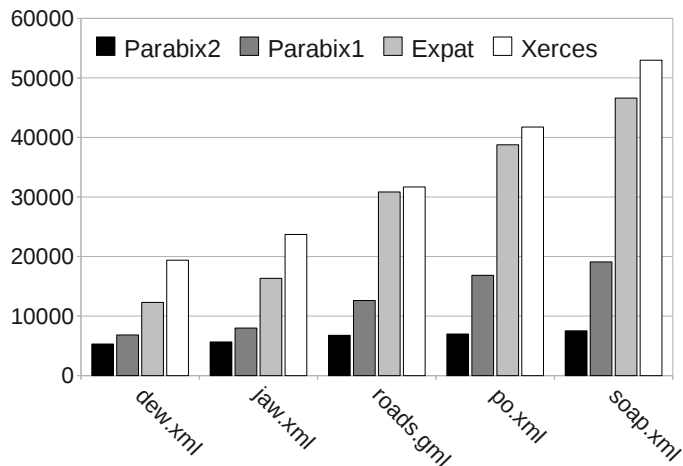


Figure 13: Core-i3 — Performance (y-axis: CPU Cycles per kB)

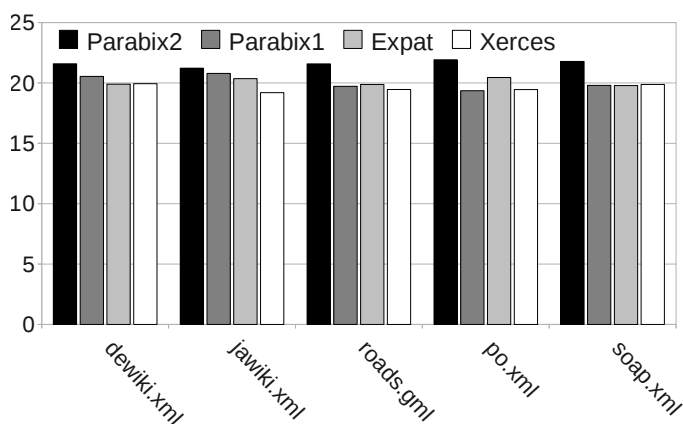


Figure 14: Core-i3 — Average Power Consumption (watts)

5.5 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 Core-i3. The average power of Core-i3 530 is about 21 watts. This Intel model has a good reputation for power efficiency. Figure 14 shows the average power consumed by each parser. Parabix2, dominated by SIMD instructions, uses approximately 5% additional power.

As shown in Figure 15, 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

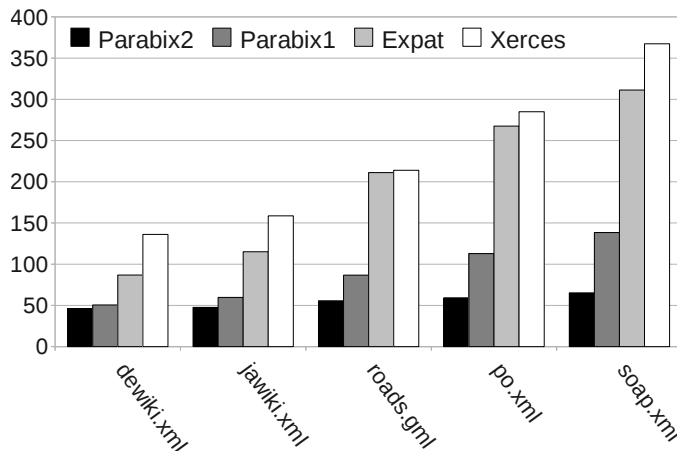


Figure 15: Core-i3 — Energy Consumption (μJ per kB)

byte respectively.

6 Scalability

6.1 Performance

Figure 16 (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 observed in migrating Parabix2 from Core2 to Core-i3; 0% to 17% performance from Core-i3 to SandyBridge. For the purpose of comparison, Figure 16 (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 Sandy-

1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139

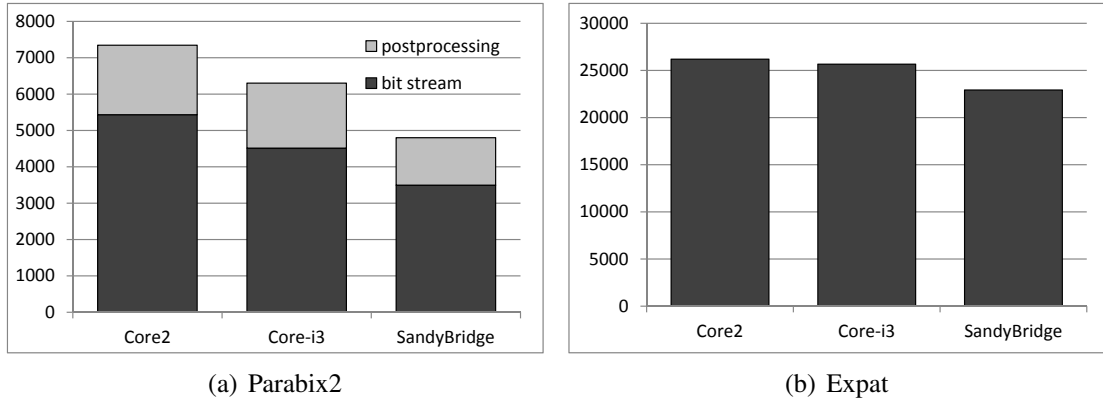


Figure 16: Average Performance Parabix vs. Expat (y-axis: CPU Cycles per kB)

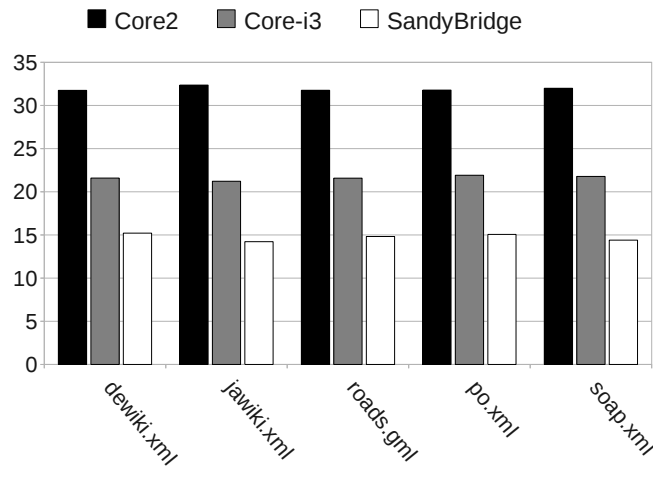


Figure 17: Average Power of Parabix2 (watts)

Bridge 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.

6.2 Power and Energy

Figure 17 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.

1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196

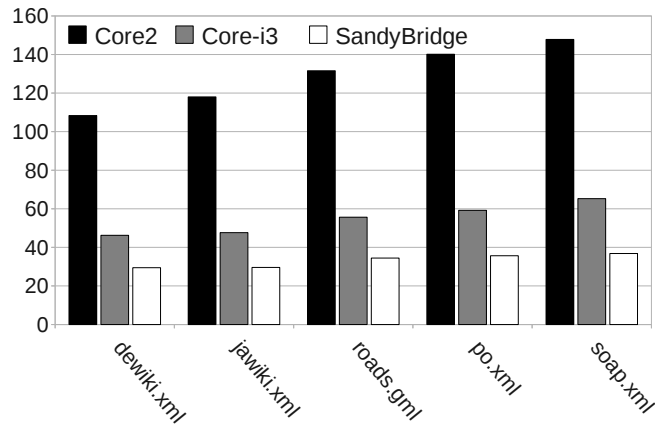


Figure 18: Energy consumption of Parabix2 (nJ/B)

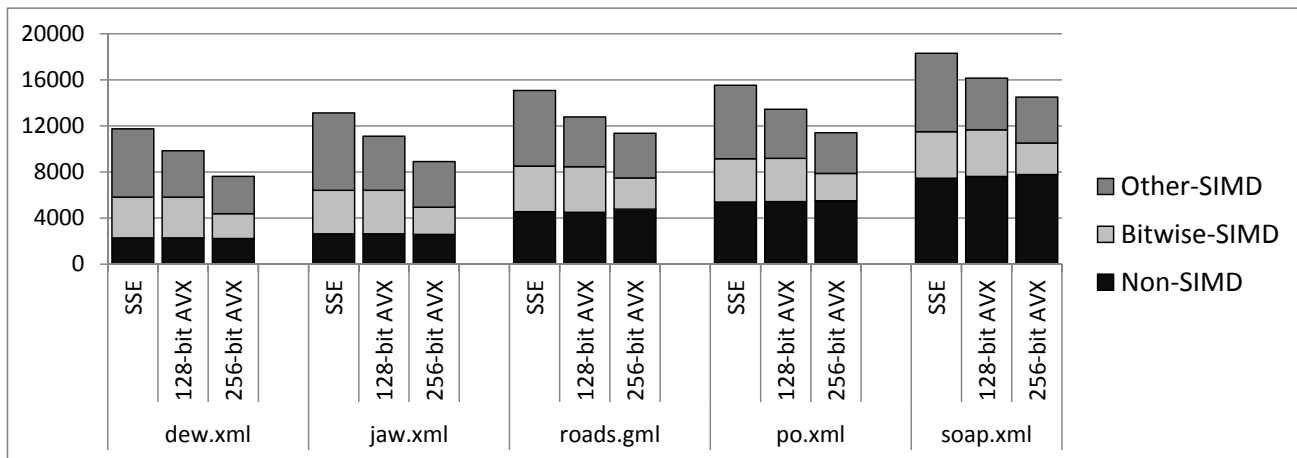


Figure 19: Parabix2 Instruction Counts (y-axis: Instructions per kB)

7 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.

7.1 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 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 ,

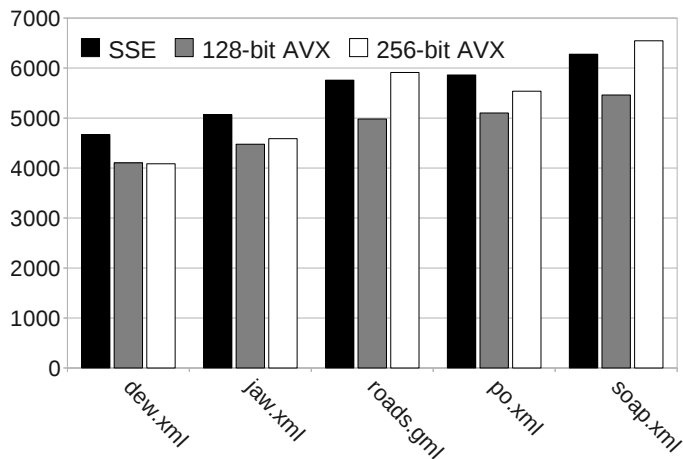


Figure 20: Parabix2 Performance (y-axis: CPU Cycles per kB)

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, $c = a [\text{op}] b$. 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.

7.2 256-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.

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 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

1254 the core library functions of Parabix2 to leverage the 256-bit AVX operations wherever possible and to
1255 simulate the remaining operations using pairs of 128-bit operations.
1256
1257

1258 Figure 19 shows the reduction in instruction counts achieved in these two versions. For each workload,
1259 the base instruction count of the Parabix2 binary compiled in SSE-only mode is shown with the caption
1260 “sse,” the version obtained by simple recompilation with AVX-mode enabled is labeled “128-bit avx,”
1261 and the version reimplemented to use 256-bit operations wherever possible is labelled “256-bit avx.” The
1262 instruction counts are divided into three classes. The “non-SIMD” operations are the general purpose
1263 instructions that use neither SSE nor AVX technology. The “bitwise SIMD” class comprises the bitwise
1264 logic operations, that are available in both 128-bit form and 256-bit form. The “other SIMD” class com-
1265 prises all other SIMD operations, primarily comprising the integer SIMD operations that are available only
1266 at 128-bit widths even with 256-bit AVX technology.
1267
1268
1269
1270
1271
1272
1273
1274
1275

1276 Note that, in each workload, the number of non-SIMD instructions remains relatively constant with each
1277 workload. As may be expected, however, the number of “bitwise SIMD” operations remains the same for
1278 both SSE and 128-bit while dropping dramatically when operating 256-bits at a time. Ideally one one may
1279 expect up to a 50% reduction in these instructions versus the 128-bit AVX. The actual reduction measured
1280 was 32%–39% depending on workload. Because some bitwise logic is needed in implementation of
1281 simulated 256-bit operations, the full 50% reduction in bitwise logic was not achieved.
1282
1283
1284
1285
1286
1287

1288 The “other SIMD” class shows a substantial 30%-35% reduction with AVX 128-bit technology com-
1289 pared to SSE. This reduction is due to eliminated copies or reloads when SIMD operations are compiled
1290 using 3-operand AVX form versus 2-operand SSE form. A further 10%–20% reduction is observed with
1291 Parabix2 version rewritten to use 256-bit operations.
1292
1293
1294
1295

1296 While the successive reductions in SIMD instruction counts are quite dramatic with the two AVX im-
1297 plementations of Parabix2, the performance benefits are another story. As shown in Figure 20, the benefits
1298 of the reduced SIMD instruction count are achieved only in the AVX 128-bit version. In this case, the ben-
1299 efits of 3-operand form seem to fully translate to performance benefits. Based on the reduction of overall
1300 Bitwise-SIMD instructions we expected a 11% improvement in performance. Instead, perhaps bizzarely,
1301 the performance of Parabix2 in the 256-bit AVX implementation does not improve significantly and actu-
1302 ally degrades for files with higher markup density (average 10%). Dewiki.xml, on which bitwise-SIMD
1303 instructions reduced by 39%, saw a performance improvement of 8%. We believe that this is primarily
1304
1305
1306
1307
1308
1309
1310

1311 due to the intricacies of the first generation AVX implementation in SandyBridge, with significant latency in
1312 many of the 256-bit instructions in comparison to their 128-bit counterparts. The 256-bit instructions also
1313 have different scheduling constraints that seem to reduce overall SIMD throughput. If these latency issues
1314 can be addressed in future AVX implementations, further substantial performance and energy benefits
1315 could be realized in XML parsing with Parabix2.
1316
1317
1318
1319
1320

1321 8 Parabix2 on GT-P1000M

1322 The Samsung Galaxy Tab GT-P1000M device houses a Samsung S5PC110 ARM Cortex-A8 single-
1323 core, dual-issue, superscalar microprocessor. In addition to the standard feature set found in such low-
1324 power 32-bit microprocessors, the S5PC110 includes the ARM NEON general-purpose SIMD engine.
1325 ARM NEON makes available a 128-bit SIMD instruction set similar in functionality to Intel SSE3 in-
1326 struction set. In this section, we present our performance comparison of a NEON-based port of Parabix2
1327 versus the Expat parser, and executed on the Samsung Galaxy Tab GT-P1000M hardware. Parabix1 and
1328 Xerces are excluded from this portion of our study due to the complexity of the cross-platform build
1329 process in porting native C/C++ applications to the Android platform.
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340

1341 8.1 Platform Hardware

1342 Samsung Galaxy Tab GT-P1000M was produced by Samsung and incorporates the ARM Cortex-A8
1343 microprocessor. Table 5 describes the Samsung Galaxy Tab GT-P1000M hardware.
1344
1345
1346

1347 Processor	ARM Cortex-A8 (1GHz)
1348 L1 Cache	32kB I-Cache, 32kB D-Cache
1349 L2 Cache	512kB
1350 Flash	16GB

1351
1352
1353
1354 Table 5: GT-P1000M

1355 8.2 Performance Results

1356 Migration of Parabix2 to the Android platform began with the retargeting of a subset of the Parabix2
1357 IDISA SIMD library for ARM NEON. This library code was cross-compiled for Android using the An-
1358 droid NDK. The Android NDK is a companion tool to the Android SDK that allows developers to build
1359 performance-critical portions of applications in native code. The majority of the Parabix2 SIMD function-
1360 ality ported directly. However, for a small subset of the SIMD functions of Parabix2 NEON equivalents
1361
1362
1363
1364
1365
1366
1367

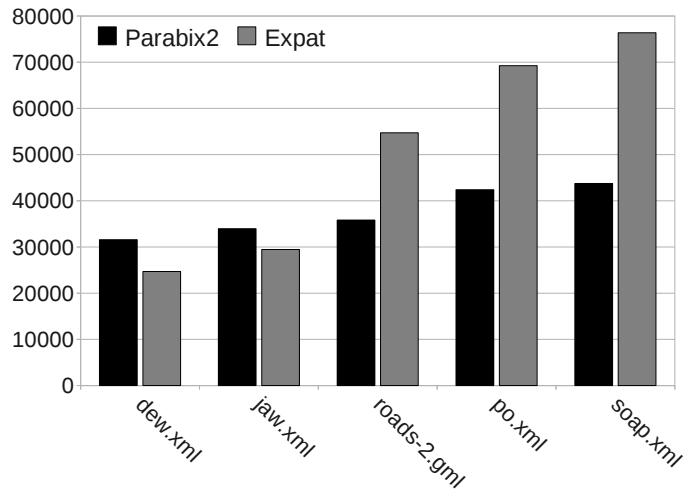


Figure 21: Parabix2 Performance on GT-P1000M (y-axis: CPU Cycles per kB)

did not exist. In such cases we simply simulated logical equivalencies using the available the instruction set.

A comparison of Figure 21 and Figure 13 demonstrates that the performance of both Parabix2 and Expat degrades substantially on Cortex-A8. This result was expected given the comparably 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.xml. 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 22 and Figure 22. 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 22 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.

1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481

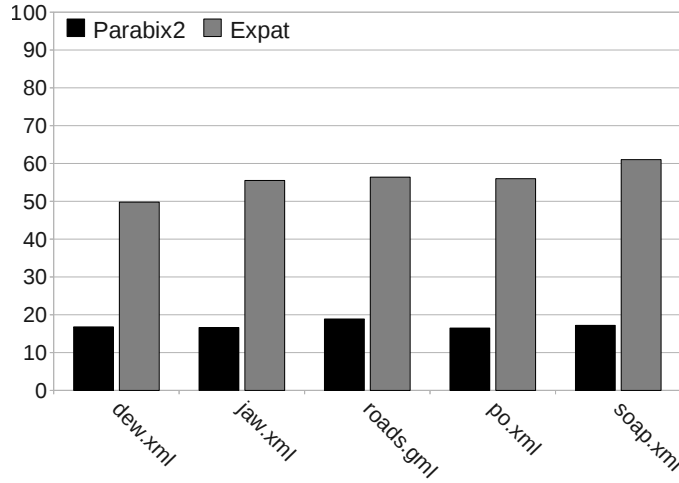


Figure 22: Relative Slow Down of Parabix2 and Expat on GT-P1000M vs. Core-i3

Stage Name	Data Structures									
	srcbuf	basis_bits	u8	lex	scope	ctCDPI	ref	tag	xml_names	check_streams
fill_buffer	write									
s2p	read	write								
classify_bytes		read		write						
validate_u8		read	write							
gen_scope				read	write					
parse_CtCDPI				read	read	write				write
parse_ref				read	read	read	write			
parse_tag				read	read	read		write		
validate_name			read	read		read	read	read	write	write
gen_check			read	read	read	read		read	read	write
postprocessing	read			read		read	read			read

Table 6: Relationship between Each Pass and Data Structures

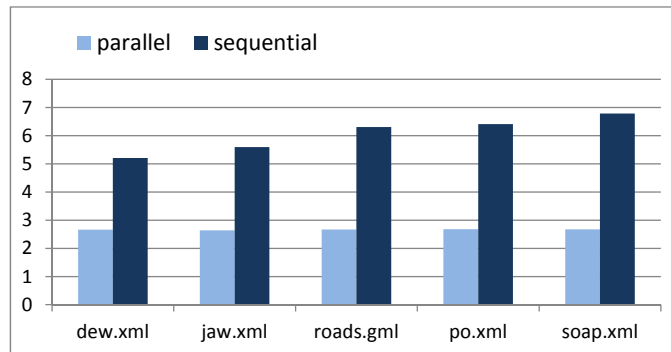


Figure 23: Processing Time (y axis: CPU cycles per byte)

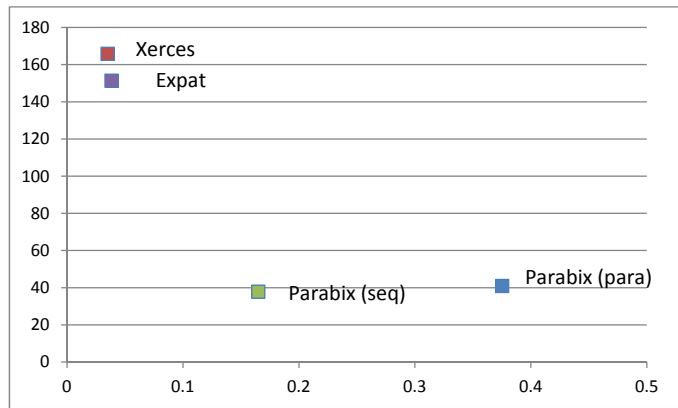


Figure 24: Energy vs. Performance (x axis: bytes per cycle, y axis: nJ per byte)

9 Multi-threaded Parabix

10 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.

References

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

- 1539 *Supercomputing*, ICS '10, pages 147–158, New York, NY, USA, 2010. ACM.
- 1540
- 1541 [3] W. Bircher and L. John. Complete system power estimation: A trickle-down approach based on performance events. In
- 1542 *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 158–168, Apr.
- 1543 2007.
- 1544 [4] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In
- 1545 *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, 2010.
- 1546 [5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fifth
- 1547 edition). W3C Recommendation, 2008.
- 1548 [6] R. Cameron, K. Herdy, and E. Amiri. Parallel bit stream technology as a foundation for XML parsing performance. In
- 1549 *International Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*, Aug. 2009.
- 1550 [7] R. D. Cameron. A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In
- 1551 *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPOPP '08,
- 1552 pages 91–98, New York, NY, USA, 2008. ACM.
- 1553 [8] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich. Parallel parsing with bitstream
- 1554 addition: An XML case study. Technical Report TR 2010-11, Simon Fraser University, School of Computing Sciece,
- 1555 October 2010.
- 1556 [9] R. D. Cameron, K. S. Herdy, and D. Lin. High performance XML parsing using parallel bit stream technology. In
- 1557 *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages
- 1558 222–235, New York, NY, USA, 2008. ACM.
- 1559 [10] J. Clark. The Expat XML Parser. <http://expat.sourceforge.net/>.
- 1560 [11] F. Corporation. Fluke Clamp Meters. <http://www.fluke.com/>.
- 1561 [12] B. DuCharme. Documents vs. data, schemas vs. schemas. In *XML 2004*, Washington D.C., 2004.
- 1562 [13] H. Esmailzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling.
- 1563 In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, 2011.
- 1564 [14] R. D. C. et. al. Parabix1. <http://parabix.costar.sfu.ca/>.
- 1565 [15] R. D. C. et. al. Parabix2. <http://parabix.costar.sfu.ca/>.
- 1566 [16] A. S. Foundation. Xerces C++ Parser. <http://xerces.apache.org/xerces-cl/>.
- 1567 [17] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz.
- 1568 Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, 2010.
- 1569 [18] 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.
- 1570 [19] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, 2010.
- 1571 [20] Y. Zhang, Y. Pan, and K. Chiu. Speculative p-DFAs for parallel XML parsing. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 388–397, Dec. 2009.
- 1572
- 1573
- 1574
- 1575
- 1576
- 1577
- 1578
- 1579
- 1580
- 1581
- 1582
- 1583
- 1584
- 1585
- 1586
- 1587
- 1588
- 1589
- 1590
- 1591
- 1592
- 1593
- 1594
- 1595