Using Parallel Bit Streams to Accelerate XML Processing

The Parabix Project

Robert D. Cameron
School of Computing Science
Simon Fraser University

President and CTO,
International Characters, Inc.

Research Seminar
IBM TJ Watson Research Center
June 5, 2008

Introduction

- Byte-at-a-time XML parsing is too slow.
  - Uses only 8 bits at a time
    - cf. 128-bit available registers and instructions.
  - Scanning loops may yield only 1 bit at a time.
    - is the next character an “<” or not?
  - XML parsing in the 100 cycles/byte range.
    - 10 MB/sec per processor GHz.
  - Can we do better?

Introduction

- Byte-at-a-time XML parsing is too slow.
  - Uses only 8 bits at a time
    - cf. 128-bit available registers and instructions.
  - Scanning loops may yield only 1 bit at a time.
    - is the next character an "<" or not?
  - XML parsing in the 100 cycles/byte range.
    - 10 MB/sec per processor GHz.
  - Can we do better?
- Parallel bit stream approach.
  - Form bit stream LAngle(i) = 1 iff byte i is "<".
  - Compute 128 bits at a time using SIMD.
  - Find next "<" with bit scan operations.
    - Built-in 32 or 64 bit operations on Intel, PPC.

The Parabix Project

- Systematic use of parallel bit streams in XML.
  - UTF-8 and XML character validation
  - UTF-8 to UTF-16 transcoding
  - Computation of lexical item streams to support parsing.
    - e.g., MarkupStart stream (for "<" or "&")
  - Parsing using bit scan operations.
  - Parallel hash value computation.
  - Parallel regular expression matching.
    - Validation of schema datatypes.
- Our current goal/bet:
  - validating XML parser at 10 cycles/byte (single core).
- Leverage bit stream parallelism for multicore.
  - Expectation: over 90% parallelizable.

Beyond Research: Open Source Technology Transfer

- Parabix is open source: parabix.costar.sfu.ca.
- Ambitious goal: be the Linux of XML middleware.
- Commitment to standards conformance, quality and portability, as well as performance.
- Commitments feed back to the research.
  - performance implications of standards proposals
  - feedback to standards activities?
- Parabix-0.53:
  - architecture for most ASCII/EBCDIC family charsets
  - DTD processing nearing completion
  - assessment with XML Conformance Test Suite underway
- SFU spin-off International Characters, Inc. is commercializing using a patentleft model.

Overview

- Part 1: SIMD notation/idealized instructions.
- Part 2: Parallel bit stream techniques
  - Fast transform to basis bit streams.
  - Character class formation.
  - Lexical item streams.
  - UTF-8 and XML character validation.
  - UTF-8 to UTF-16 transcoding.
  - Parallel regular expression matching.
- Part 3: Parabix Performance Study
  - Parabix 0.53 vs. Expat, Xerces
- Part 4: Performance Prospects.
  - single core, multicore
- Conclusions

SIMD Notation

- An idealized SIMD notation simplifies and provides portability
  - SSE, MMX
  - Altivec/Cell PPE, SPE
- r = simd_op/w(r1, r2)
  - simultaneous application of operation op to all fields of width w

SIMD Notation

- An idealized SIMD notation simplifies and provides portability
  - SSE, MMX
  - Altivec/Cell PPE, SPE
- r = simd_op/w(r1, r2)
  - simultaneous application of operation op to all fields of width w
- r = simd_add/8(r1, r2)
  - partition r, r1 and r2 into 8-bit fields
  - add corresponding 8-bit fields of r1 and r2 to produce fields of r

Inductive Doubling Support

- The notation also provides systematic support for inductive doubling:
  - algorithms that repeatedly double field widths or other data attributes
- SIMD operations defined for all field widths w = 2, 4, 8, ...
- Half-operand modifiers may be applied to input operands to select either the high (h) or low (l) w/2 bits of each field
- Note to chip architects: implementing our inductive doubling instruction set architecture would speed up many algorithms (ours and others)!

Inductive Doubling Example

- Example: compute population count of each 16-bit field of rA → rB
- Add the low bit of each 2-bit field to the high bit.
  `t1 = simd_add/2(rA/l, rA/h)`
- We now have 64 2-bit sums.

Inductive Doubling Example

- Example: compute population count of each 16-bit field of rA → rB
- Add the low bit of each 2-bit field to the high bit.
  `t1 = simd_add/2(rA/l, rA/h)`
- We now have 64 2-bit sums.
- Combine the low and high 2-bit sums in 4-bit fields.
  `t2 = simd_add/4(t1/l, t1/h)`

Inductive Doubling Example

- Example: compute population count of each 16-bit field of rA → rB
- Add the low bit of each 2-bit field to the high bit.
  `t1 = simd_add/2(rA/l, rA/h)`
- We now have 64 2-bit sums.
- Combine the low and high 2-bit sums in 4-bit fields.
  `t2 = simd_add/4(t1/l, t1/h)`
- Combine the low and high 4-bit sums in 8-bit fields.
  `t3 = simd_add/8(t2/l, t2/h)`

Inductive Doubling Example

- Example: compute population count of each 16-bit field of rA → rB
- Add the low bit of each 2-bit field to the high bit.
  t1 = simd_add/2(rA/l, rA/h)
- We now have 64 2-bit sums.
- Combine the low and high 2-bit sums in 4-bit fields.
  t2 = simd_add/4(t1/l, t1/h)
- Combine the low and high 4-bit sums in 8-bit fields.
  t3 = simd_add/8(t2/l, t2/h)
- Now combine the 8-bit sums for 16-bit pop count.
  rB = simd_add/16(t3/l, t3/h)

Transposition to Parallel Bit Streams

- Start with 8 consecutive registers s0, s1, s2, ... s7 of serial byte data.
- Produce 8 parallel registers of serial bit stream data p0, p1, ..., p7.
- Three stage algorithm:
  - produce 2 streams of serial nybble data
  - then 4 streams of serial bitpair data
  - finally 8 streams of serial bit data
- Uses simd_pack: r = simd_pack/w(a,b)
  - convert each w-bit field of a and b to w/2 bits and pack them together consecutively

Idealized Transposition Stages

- High nybble stream (½ of stage 1)
  - pack high 4 bits of each consecutive pair of 8-bit fields.

```
b0123_0 = simd_pack/8(s0/h, s1/h)
b0123_1 = simd_pack/8(s2/h, s3/h)
b0123_2 = simd_pack/8(s4/h, s5/h)
b0123_3 = simd_pack/8(s6/h, s7/h)
```

Idealized Transposition Stages

- High nybble stream (½ of stage 1)
  - pack high 4 bits of each consecutive pair of 8-bit fields.

```
b0123_0 = simd_pack/8(s0/h, s1/h)
b0123_1 = simd_pack/8(s2/h, s3/h)
b0123_2 = simd_pack/8(s4/h, s5/h)
b0123_3 = simd_pack/8(s6/h, s7/h)
```

- Bits 2/3 bitpair stream (¼ of stage 2)
  - pack low 2 bits of each consecutive pair of high nybbles.

```
b23_0 = simd_pack/4(b0123_0/l, b0123_1/l)
b23_1 = simd_pack/4(b0123_2/l, b0123_3/l)
```

Idealized Transposition Stages

- High nybble stream (½ of stage 1)
  - pack high 4 bits of each consecutive pair of 8-bit fields.

```
b0123_0 = simd_pack/8(s0/h, s1/h)
b0123_1 = simd_pack/8(s2/h, s3/h)
b0123_2 = simd_pack/8(s4/h, s5/h)
b0123_3 = simd_pack/8(s6/h, s7/h)
```

- Bits 2/3 bitpair stream (¼ of stage 2)
  - pack low 2 bits of each consecutive pair of high nybbles.

```
b23_0 = simd_pack/4(b0123_0/l, b0123_1/l)
b23_1 = simd_pack/4(b0123_2/l, b0123_3/l)
```

- Bit 2 and 3 bitstreams (¼ of stage 3)

```
bit2 = simd_pack/2(b23_0/h, b23_1/h)
bit3 = simd_pack/2(b23_0/l, b23_1/l)
```

Transposition Summary

- Idealized transposition requires 3 stages of 8 operations each.
- Using 128-bit registers: transpose 128 bytes in 24 operations.
- Runs on SSE, Altivec, SPE with idealized library..
- Better Altivec/SSE algorithms based on pack/16; Altivec: 72 ops/128 bytes.
- Future: CPU support for single-cycle idealized instructions => transposition at 0.2 cycles/byte.
  - Attention chip architects!

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream
- compile(`[CharDef(LAngle, “<”)]`)

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream
- compile([CharDef(LAngle, “<”)])

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_and(bit[2], bit[3]);
temp3 = simd_andc(temp2, temp1);
temp4 = simd_and(bit[4], bit[5]);
temp5 = simd_or(bit[6], bit[7]);
temp6 = simd_andc(temp4, temp5);
LAngle = simd_and(temp3, temp6);
```

- 7 operations per 128 characters.

Multiple Class Formation

- Common subexpression simplify.
- compile([CharDef(LAngle, “<”),
)

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_and(bit[2], bit[3]);
temp3 = simd_andc(temp2, temp1);
temp4 = simd_and(bit[4], bit[5]);
temp5 = simd_or(bit[6], bit[7]);
temp6 = simd_andc(temp4, temp5);
LAngle = simd_and(temp3, temp6);
```

Multiple Class Formation

- Common subexpression simplify.
- compile([CharDef(LAngle, “<”),
  CharDef(RAngle, “>”)])

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_and(bit[2], bit[3]);
temp3 = simd_andc(temp2, temp1);
temp4 = simd_and(bit[4], bit[5]);
temp5 = simd_or(bit[6], bit[7]);
temp6 = simd_andc(temp4, temp5);
LAngle = simd_and(temp3, temp6);
temp7 = simd_andc(bit[6], bit[7]);
temp8 = simd_and(temp4, temp7);
RAngle = simd_and(temp3, temp8);
```

Character Ranges

- Ranges may require fewer operations!
- compile(`[CharSet('Control', ['\x00-\x1F']),`
  `)`

Character Ranges

- Ranges may require fewer operations!
- compile(`[CharSet('Control', ['\x00-\x1F']),`
)

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_or(temp1, bit[2]);
Control = simd_andc(simd_const_1(1), temp2);
```

Character Ranges

- Ranges may require fewer operations!
- compile([CharSet('Control', ['\x00-\x1F']),
  CharSet('Digit', ['0-9'])])

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_or(temp1, bit[2]);
Control = simd_andc(simd_const_1(1), temp2);
```

Character Ranges

- Ranges may require fewer operations!
- compile([CharSet('Control', ['\x00-\x1F']),
  CharSet('Digit', ['0-9'])])

```
temp1 = simd_or(bit[0], bit[1]);
temp2 = simd_or(temp1, bit[2]);
Control = simd_andc(simd_const_1(1), temp2);
temp3 = simd_and(bit[2], bit[3]);
temp4 = simd_andc(temp3, temp1);
temp5 = simd_or(bit[5], bit[6]);
temp6 = simd_and(bit[4], temp5);
Digit = simd_andc(temp4, temp6);
```

Lexical Item Streams

- Using the character class compiler, we define a set of lexical item streams for XML.
  - MarkupStart, NameFollow, WhiteSpace, QuoteScan, Hyphen, Qmark, CDend, Hex, Digit
  - 67 operations in total to classify 128 bytes.
  - 0.5 ops per byte.
- Can define with no interblock dependencies
  - data parallel distribution to multiple cores.

UTF-8 Byte Classification

- UTF-8 bytes are single-byte sequences, or prefixes or suffixes of multibyte sequences.
- Classify 128 at a time.

```
u8unibyte = simd_not(u8bit0);
u8prefix = simd_and(u8bit0, u8bit1);
u8suffix = simd_andc(u8bit0, u8bit1);
u8prefix2 = simd_andc(u8prefix, u8bit2);
u8pfx3or4 = simd_and(u8prefix, u8bit2);
u8prefix3 = simd_andc(u8pfx3or4,u8bit3);
u8prefix4 = simd_and(u8pfx3or4, u8bit3);
```

- 7 cycles/128 bytes.

UTF-8 Scope Streams

- Identify suffix expectations for prefix bytes.
- Shift forward logical immediate of 1-3 positions.
  - (forward = left for big-endian, right for little-endian)

```
scope22 = simd_sfli(u8prefix2, 1);
   ...
scope43 = simd_sfli(u8prefix4, 2);
scope44 = simd_sfli(u8prefix4, 3);
s_nn = simd_or(simd_or(scope22,scope33),
               scope44);
any = simd_or(simd_or(scope32, scope42),
                  simd_or(scope43, s_nn));
```

- 6 shifts, 5 logic ops/128 bytes.

UTF-8 Validation

- Suffixes must occur where expected.

```
err_mask = simd_xor(any, u8suffix);
```

- Prefix bytes 0xC0, 0xC1 are illegal.

```
C0C1= simd_andc(u8prefix2,
        simd_or(simd_or(u8bit3, u8bit4),
                simd_or(u8bit5, u8bit6));
err_mask = simd_or(err_mask, C1);
```

- Other constraints similar.
- 26 logic and 4 shift operations for validation.

Transcoding to UTF-16

- XML files typically stored in UTF-8
  - variable-length byte-oriented encoding
- Applications often use UTF-16 internally
  - fixed 16-bits per character (except rare characters in supplementary plane)
- UTF-8 to UTF-16 transcoding a typical requirement for XML parsers.
- Frequently cited as a major cost: 30% or more.

Transcoding to UTF-16 (cont'd)

- Calculate 16 parallel bit streams using logic and shift operations.
  - About 4 ops per bit stream per block.
- Principal challenge: variable length mapping
  - Every 1, 2 or 3 byte UTF-8 seq.: 1 UTF-16 value.
  - 4-byte UTF-8 sequences: 2 UTF-16 values.
  - Convention: calculate UTF-16 bit values at position of last byte in sequence (& scope42).
  - Output is only generated for these positions.
  - Mapping is achieved by parallel bit deletion.

Parallel Bit Deletion

- Mark all positions to be deleted.

```
delmask = simd_or(u8prefix, scope32, scope43)
```

- Apply a parallel deletion algorithm.
- Ideal algorithm: deletion by central induction.
  - Move bits to center within each field.
  - Solve 4-bit fields, then 8-bit, then 16 ...
  - Use SIMD rotate of PPU/SPU.
  - One rotate per field width per stream.

UTF-8 to UTF-16 on Cell

- UTF-8 to UTF-16 transcoding has been implemented and tested on PPE.
- Distribution to SPEs involves same transition boundary issues as UTF-8 validation.
- SPE implementation should benefit from greater register availability:
  - eliminate PPE loads and stores for temporary values due to register pressure

Regular Expression Matching

- Parallel Matching of [0-9]* Regular Expression
  - Match 5 instances starting from 5 cursors

```
NaN  43215   594356   211 token     character stream
00000111110001111110001110000000   [0-9] character class
00100000010000000100000100000 1   c0, initial cursor
```

Regular Expression Matching

- Parallel Matching of [0-9]* Regular Expression
  - Match 5 instances starting from 5 cursors
  - Add the bitstreams!

```
NaN  43215   594356   211 token      character stream
00000111110001111110001110000000    [0-9] character class
00100000010000000100000100000100    c0, initial cursor
00101000000010000000100000000100    c0 + [0-9]
```

Regular Expression Matching

- Parallel Matching of [0-9]* Regular Expression
  - Match 5 instances starting from 5 cursors
  - Add the bitstreams!

```
NaN  43215   594356   211 token      character stream
00000111110001111110001110000000    [0-9] character class
00100000010000000100000100000 1     c0, initial cursor
00101000000010000000100000000 1     c0 + [0-9]
```

  - Carry propagation moves the cursors through all matching [0-9] characters!

Regular Expression Matching

- Matching [-+]? (zero or one sign)

```
NaN  4321-   59435+   211   c++      character stream
000000000100000000100000000011      [-+] character class
001000000100000000100000100001      c0, initial cursor
```

Regular Expression Matching

- Matching [-+]? (zero or one sign)
  - Limit propagation by masking.

`NaN  4321-  59435+  211  c++`	character stream
`000000000100000000100000000011`	[-+] character class
`001000000100000000100000100001`	c0, initial cursor
`000000000100000000100000000001`	c0 & [-+]

Regular Expression Matching

- Matching [-+]? (zero or one sign)
  - Limit propagation by masking.
  - Add the bitstreams!

`NaN  4321-  59435+  211  c++`	character stream
`000000000**1**00000000**1**0000000000**11**`	[-+] character class
`00**1**000000**1**00000000**1**00000**1**000000**1**`	c0, initial cursor
`000000000**1**00000000**1**0000000000**1**`	c0 & [-+]
`00**1**00000**1**00000000**1**000000**1**0000**1**0`	c0 + (c0 & [-+])

Composite Expression Matching

`^[-+]?[0-9]+$` (signed integers anchored at each end)

`;5.796953 - 6++ 4+ gnorw 17- 421`	character stream
`0000000000**1**00**11**00**1**0000000000**1**0000`	[-+] character class
`0**1**0**1111111**000**1**000**1**0000000**11**00**111**`	[0-9] character class
`0000000**1**0**1**000**1**00**1**00000**1**000**1**0001`	**c0, initial cursor**
`**1**0000000**1**0**1**000**1**00**1**00000**1**0001000`	**end_mask**

Composite Expression Matching

`^[-+]?[0-9]+$` (signed integers anchored at each end)

`;5.796953 - 6++ 4+ gnorw 17- 421`	character stream
`0000000000**1**00**11**00**1**0000000000**1**0000	[-+] character class
`0**1**0**1111111**000**1**000**1**0000000**11**00**111**	[0-9] character class
`0000000**1**0**1**000**1**00**1**00000**1**000**1**0001	**c0, initial cursor**
10000000**1**0**1**000**1**00**1**00000**1**0001000	**end_mask**
`0000000**11**000**1**00**1**00000**1**00**1**00001	**c1 = c0 + (c0 & [-+])**

Composite Expression Matching

`^[-+]?[0-9]+$` (signed integers anchored at each end)

`;5.796953 - 6++ 4+ gnorw 17- 421`	character stream
`0000000000**1**00**11**00**1**0000000000**1**0000	[-+] character class
`0**1**0**1111111**000**1**000**1**0000000**11**00**111**	[0-9] character class
`0000000**1**0**1**000**1**00**1**00000**1**000**1**0001	**c0, initial cursor**
`**1**0000000**1**0**1**000**1**00**1**00000**1**000**1**000	**end_mask**
`0000000**11**000**1**00**1**00000**1**00**1**00001	**c1 = c0 + (c0 & [-+])**
`00**1**0000000000000**1**0000000**1**000**1**000	**(c1+[0-9])&~[0-9] &~c1**

Composite Expression Matching

`^[-+]?[0-9]+$` (signed integers anchored at each end)

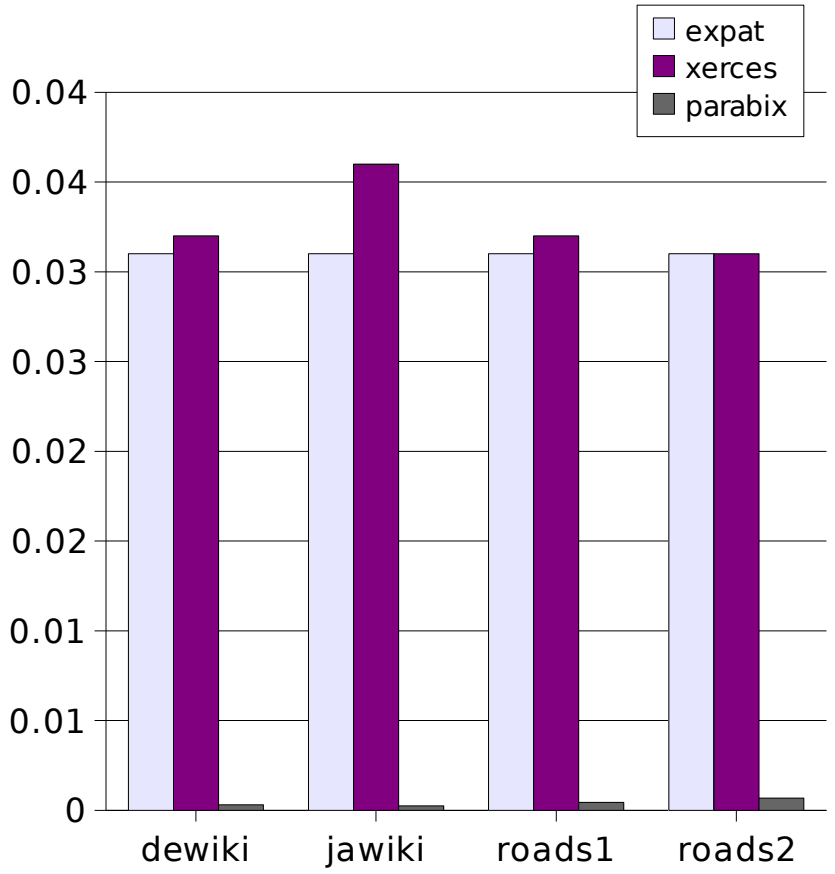
`;5.796953 - 6++ 4+ gnorw 17- 421`	character stream
`0000000000**1**00**11**00**1**0000000000**1**0000	[-+] character class
`0**1**0**1111111**000**1**000**1**0000000**11**00**111**	[0-9] character class
`0000000**1**0**1**000**1**00**1**00000**1**000**1**0001	**c0, initial cursor**
10000000**1**0**1**000**1**00**1**00000**1**0001000	**end_mask**
`0000000**11**000**1**00**1**00000**1**00**1**00001	**c1 = c0 + (c0 & [-+])**
`00**1**0000000000000**1**0000000**1**0001000	**(c1+[0-9])&~[0-9] &~c1**
`0000000000000000**1**0000000**1**0001000	**end_mask & c2**

Three complete matches found.

Parabix Performance Study

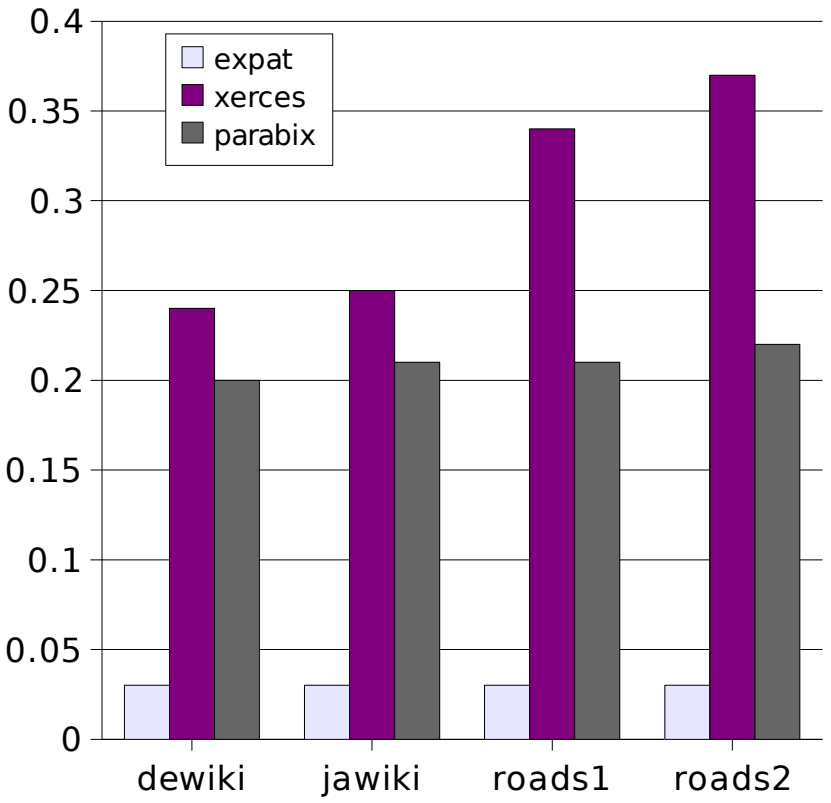
- Parabix vs. Expat, Xerces-C (SAX)
- Use markup statistics application.
- Use PAPI performance counters.
  - L1 and L2 cache misses
  - Conditional branches; mispredications
  - Instruction counts
  - Cycles per byte
- Sample data:
  - 2 text-oriented files: German, Japanese
  - 2 data-oriented files: small, large GML

L2 Data Cache Misses Per Byte



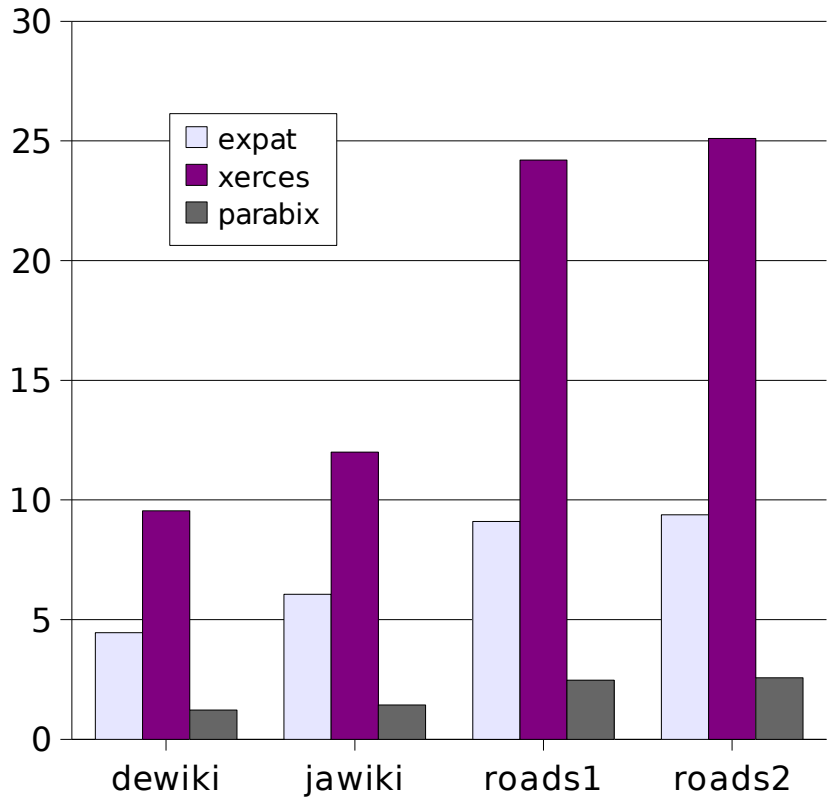
Parabix has excellent L2 cache behaviour.

L1 Data Cache Misses Per Byte



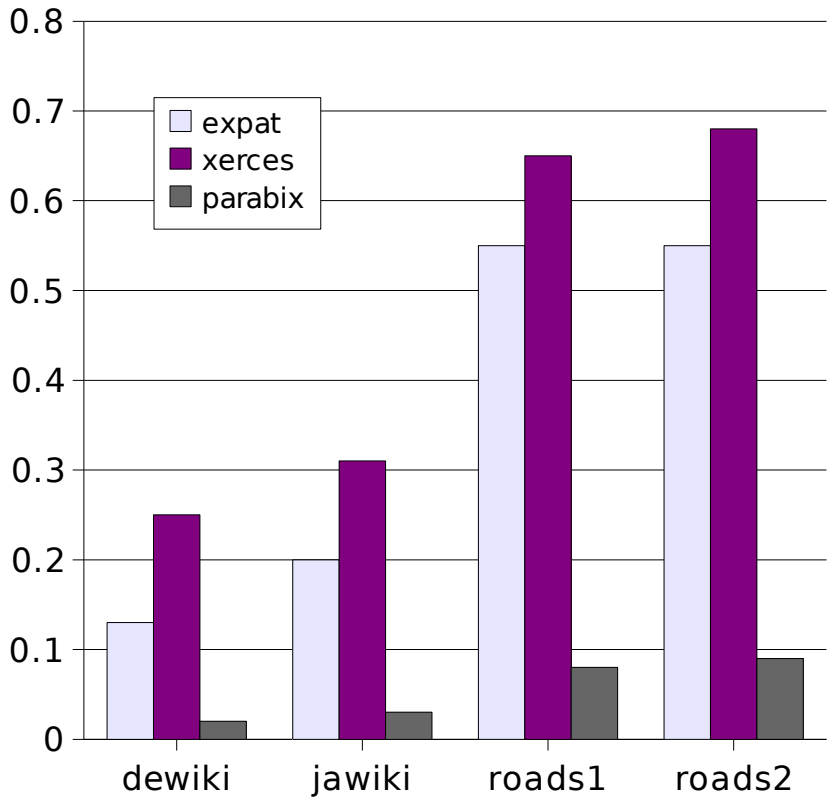
L1 cache behaviour is an area for further work.

Conditional Branches Per Byte



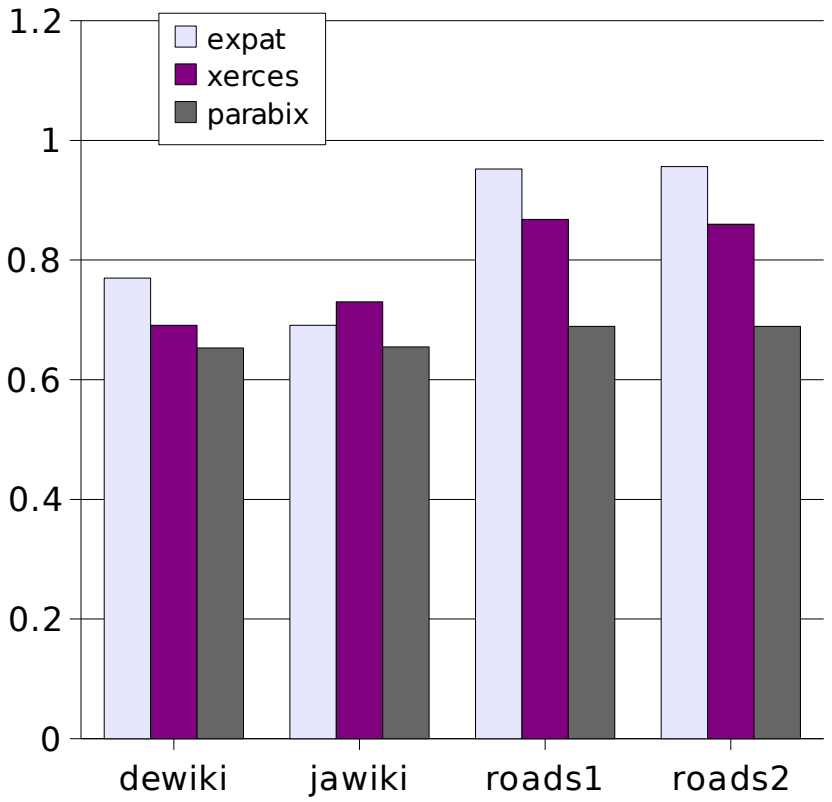
Far fewer branches in parallel bit stream code.

Branch Mispredications Per Byte



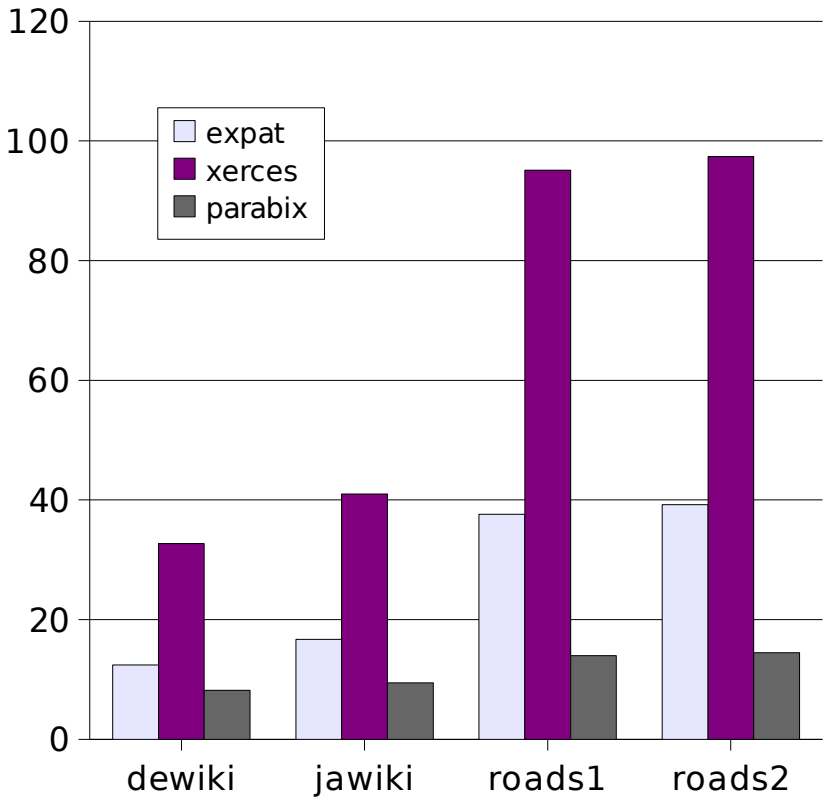
Far fewer branch mispredictions.

Cycles Per Instruction

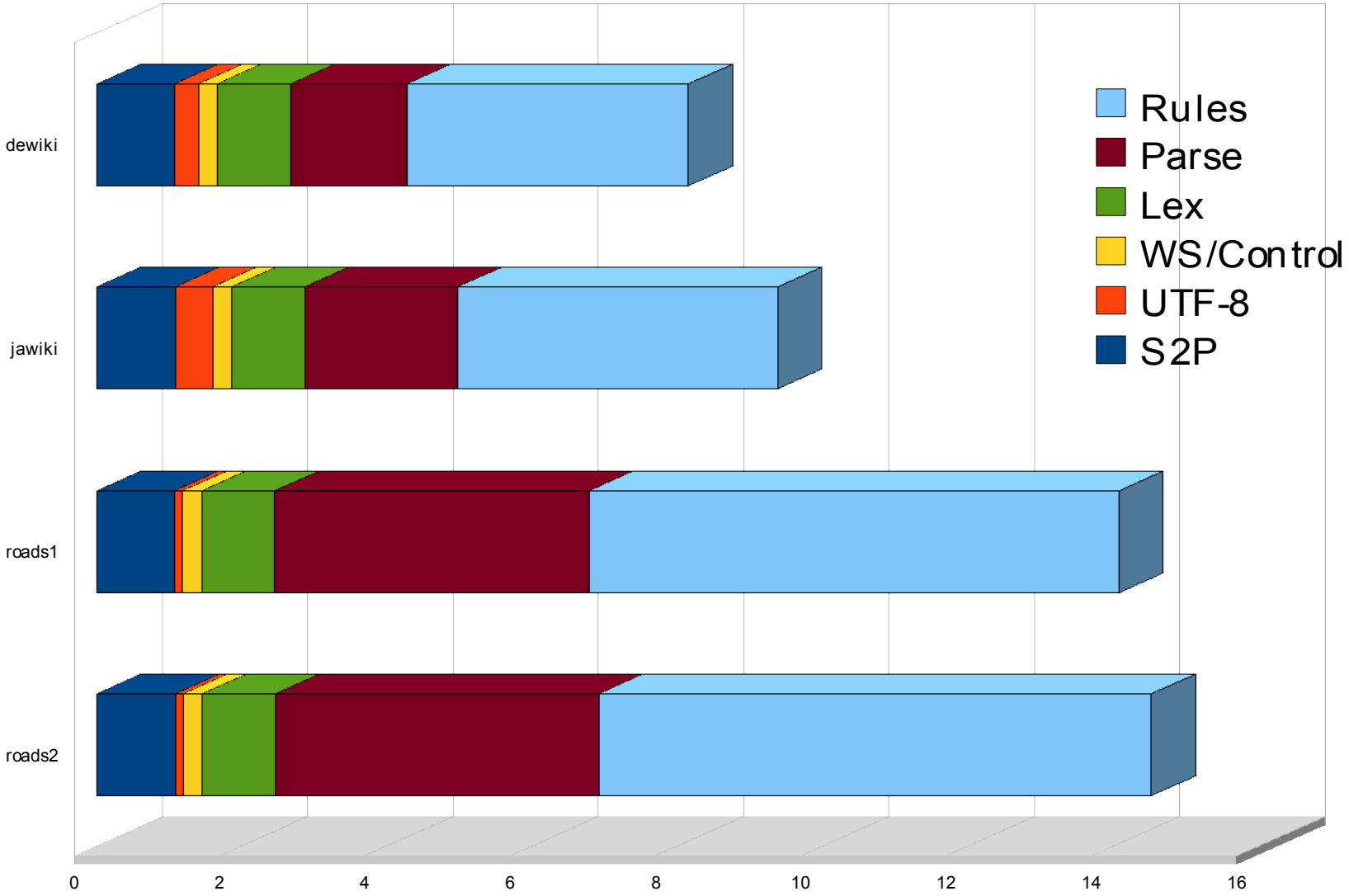


Better utilization of processor resources.

CPU Cycles Per Byte



Parabix Component Performance (Cycles Per Byte)



Performance Notes

- Parallel bit stream components perform well
  - S2P, UTF8/XML validation, WS/Control, Lexical Items
  - Less than 3 cycles/byte.
- Parser proper is < 5 cycles/byte.
  - Inherently sequential
  - Branches after each scan
  - Difficult to partition
- Symbol table/well-formedness rules
  - Use STL hashmaps throughout.
  - Not parallelized.
  - Major performance bottleneck at present.

Performance Prospects

- Parallel bit stream components
  - Some further optimization
  - Inductive doubling 3X speedups: S2P, || deletion
  - Data parallel distribution to multicore straightforward.
    - small overlap for UTF-8 sequences at partition boundaries.
- Develop fast Comment/PI/CDATA preparser.
  - Mask off contents from lexical streams
  - Remaining “<” and “&” must be markup.
  - Independently parse complete markups within partitions.
- Symbol table/semantics
  - Use length-sorted multipass symbol lookup.
    - initial results: 2X improvement
  - parallel hash value computation
  - XML Screamer techniques: schema compilation

Conclusions

- Parallel bit stream technology offers dramatic performance improvements for XML and other text applications.
- Performance improvements can be demonstrated in real-world application.
- Intraregister parallelism can be leveraged for intrachip parallelism (multicore).
- Parabix is open source.