Parallel Bit Stream Technology as a Foundation for XML Parsing Performance

Rob Cameron, Ken Herdy and Ehsan Amiri

School of Computing Science
Simon Fraser University

International Characters, Inc.

August 10, 2009

Outline

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.
- But what about the bulk of the world's XML processing?

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.
- But what about the bulk of the world's XML processing?
- Our expectation: commodity Intel/AMD/Power PC chips will continue to dominate.

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.
- But what about the bulk of the world's XML processing?
- Our expectation: commodity Intel/AMD/Power PC chips will continue to dominate.
- Our work: systematic exploitation of SIMD capabilities of these chips for XML processing.

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.
- But what about the bulk of the world's XML processing?
- Our expectation: commodity Intel/AMD/Power PC chips will continue to dominate.
- Our work: systematic exploitation of SIMD capabilities of these chips for XML processing.
- Our innovation: parallel bit stream technology.

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.
- But what about the bulk of the world's XML processing?
- Our expectation: commodity Intel/AMD/Power PC chips will continue to dominate.
- Our work: systematic exploitation of SIMD capabilities of these chips for XML processing.
- Our innovation: parallel bit stream technology.
  - A method to process 128 bytes at a time using 128-bit SIMD registers.

XML on Commodity Processors

- XML appliances and chips have important enterprise applications.
- But what about the bulk of the world's XML processing?
- Our expectation: commodity Intel/AMD/Power PC chips will continue to dominate.
- Our work: systematic exploitation of SIMD capabilities of these chips for XML processing.
- Our innovation: parallel bit stream technology.
  - A method to process 128 bytes at a time using 128-bit SIMD registers.
- Results: exceptionally promising.

A Transform Representation of Text

- Given a byte-oriented character stream T.

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

T	A	b	1	7	;

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

T	A	b	1	7	;
ASCII	01000001	01100010	00110001	00110111	00111011

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

T	A	b	1	7	;
ASCII	01000001	01100010	00110001	00110111	00111011
b_0	0	0	0	0	0

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

T	A	b	1	7	;
ASCII	01000001	01100010	00110001	00110111	00111011
b_0	0	0	0	0	0
b_1	1	1	0	0	0

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

T	A	b	1	7	;
ASCII	01000001	01100010	00110001	00110111	00111011
b_0	0	0	0	0	0
b_1	1	1	0	0	0
b_2	0	1	1	1	1

A Transform Representation of Text

- Given a byte-oriented character stream T.
- Transpose to 8 parallel bit streams b_0, b_1, ..., b_7.
- Each stream b_k comprises bit k of each byte of T.

T	A	b	1	7	;
ASCII	01000001	01100010	00110001	00110111	00111011
b_0	0	0	0	0	0
b_1	1	1	0	0	0
b_2	0	1	1	1	1
b_3	0	0	1	1	1
b_4	0	0	0	0	1
b_5	0	0	0	1	0
b_6	0	1	0	1	1
b_7	1	0	1	1	1

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".
  - Byte-at-time loop computes only 1 bit per iteration!

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".
  - Byte-at-time loop computes only 1 bit per iteration!
- So let's compute those bits in parallel!

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".
  - Byte-at-time loop computes only 1 bit per iteration!
- So let's compute those bits in parallel!
  - Bitwise logic on basis streams $b_i \longrightarrow$ [<] stream.

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".
  - Byte-at-time loop computes only 1 bit per iteration!
- So let's compute those bits in parallel!
  - Bitwise logic on basis streams $b_i \longrightarrow$ [<] stream.
  - Process 128 positions at a time using SSE registers.

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".
  - Byte-at-time loop computes only 1 bit per iteration!
- So let's compute those bits in parallel!
  - Bitwise logic on basis streams $b_i \longrightarrow$ [<] stream.
  - Process 128 positions at a time using SSE registers.
- Find next "<" with bit scan instruction (BSF).

High Performance Text Processing

Why form parallel bit streams?

- Byte-at-a-time text processing is too slow.
  - Example: XML scan for "<".
  - Byte-at-time loop computes only 1 bit per iteration!
- So let's compute those bits in parallel!
  - Bitwise logic on basis streams $b_i \longrightarrow$ [<] stream.
  - Process 128 positions at a time using SSE registers.
- Find next "<" with bit scan instruction (BSF).
  - Advance up to 63 positions at once.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

CASCON '08: XML statistics application

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

CASCON '08: XML statistics application

- Xerces C: 32-143 CPU cycles/byte.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

CASCON '08: XML statistics application

- Xerces C: 32-143 CPU cycles/byte.
- Expat: 14-58 CPU cycles/byte.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

CASCON '08: XML statistics application

- Xerces C: 32-143 CPU cycles/byte.
- Expat: 14-58 CPU cycles/byte.
- Parabix: 5-14 CPU cycles/byte.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

CASCON '08: XML statistics application

- Xerces C: 32-143 CPU cycles/byte.
- Expat: 14-58 CPU cycles/byte.
- Parabix: 5-14 CPU cycles/byte.
- 10X fewer L2 data cache misses.

Reported Results

PPoPP '08: UTF-8 to UTF-16 transcoding

- iconv: 17.6-23.2 CPU cycles/byte.
- u8u16: 0.9 - 6.8 CPU cycles/byte.
- Average case speedup: about 10X.

CASCON '08: XML statistics application

- Xerces C: 32-143 CPU cycles/byte.
- Expat: 14-58 CPU cycles/byte.
- Parabix: 5-14 CPU cycles/byte.
- 10X fewer L2 data cache misses.
- 10X fewer branch mispredictions.

Reported Results

SVG Open '08: GML to SVG Conversion

- JAXP/SAX with Xerces-J: 220 CPU cycles/byte.
- JAXP/SAX with Crimson: 230 CPU cycles/byte.
- JAXP/SAX with Intel XSS: 210 CPU cycles/byte.
- JAXP/XSLT with Saxon: 155 CPU cycles/byte.
- JAXP/XSLT with Intel XSS: 65 CPU cycles/byte.
- C++/SAX with Intel XSS: 25 CPU cycles/byte.
- C++/ILAX with Parabix: 15 CPU cycles/byte.

Performance Prospects

- First generation Parabix has known bottlenecks.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.
  - Symbol lookup: 50% of XML statistics application.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.
  - Symbol lookup: 50% of XML statistics application.
- Improved techniques identified in both areas.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.
  - Symbol lookup: 50% of XML statistics application.
- Improved techniques identified in both areas.
  - Parallel parsing with bitstream addition.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.
  - Symbol lookup: 50% of XML statistics application.
- Improved techniques identified in both areas.
  - Parallel parsing with bitstream addition.
  - Length-sorted lookup to eliminate loops.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.
  - Symbol lookup: 50% of XML statistics application.
- Improved techniques identified in both areas.
  - Parallel parsing with bitstream addition.
  - Length-sorted lookup to eliminate loops.
- Bit stream parallelism may be leveraged for intrachip parallelism on multicore processors.

Performance Prospects

- First generation Parabix has known bottlenecks.
  - Sequential parsing: 25% of XML statistics application.
  - Symbol lookup: 50% of XML statistics application.
- Improved techniques identified in both areas.
  - Parallel parsing with bitstream addition.
  - Length-sorted lookup to eliminate loops.
- Bit stream parallelism may be leveraged for intrachip parallelism on multicore processors.
- Processor advances amplify the advantage of parallel bit stream methods.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]
- Length-sorted fast symbol lookup.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]
- Length-sorted fast symbol lookup.
- Java performance: Array Set Model.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]
- Length-sorted fast symbol lookup.
- Java performance: Array Set Model.
- Compiler technology.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]
- Length-sorted fast symbol lookup.
- Java performance: Array Set Model.
- Compiler technology.
  - Program using high-level unbounded bitstream logic.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]
- Length-sorted fast symbol lookup.
- Java performance: Array Set Model.
- Compiler technology.
  - Program using high-level unbounded bitstream logic.
  - Compile to low-level SIMD code.

Parabix 2

Current focus of work: Parabix 2 Project.

- Full catalog of parallel bitstream techniques.
- Parallel parsing with bitstream addition.
- Targetting for new/prospective SIMD architectures.
  - Intel SSE 4.1, 4.2, AMD SSE 5
  - Intel AVX (256-bit)
  - GPGPUs: e.g., Intel Larrabee
  - Future: pex/pdep, inductive doubling ISA [ASPLOS '09]
- Length-sorted fast symbol lookup.
- Java performance: Array Set Model.
- Compiler technology.
  - Program using high-level unbounded bitstream logic.
  - Compile to low-level SIMD code.
  - Design for multiple back-ends.

Bit Stream Types

Our catalog of XML bit streams shows how bit streams can be used in many ways.

- Basis bit streams.
- Character class bit streams.
- Scope streams.
- Multiliteral bit streams.
- Lexical item streams.
- Error streams.
- Deletion mask streams.
- Cursor Streams.
- Call-out Streams.

Parallel Scanning Basics

Parallel Scanning Basics

- Given a text with numeric references, T

T `12
&#10: deed 3443 ⡩ (&#8;)`

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)

T `);8#&( ;54301#& 3443 deed :01#&;3100#& 21`

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.

```
T   );8#&( ;54301#& 3443 deed :01#&;3100#& 21
D   ..1.....11111...1111.......11...1111...11
```

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.
- Initialize cursors C_0 using the '&#' stream .

```
T   );8#&( ;54301#& 3443 deed :01#&;3100#& 21
D   ..1.....11111...1111.......11...1111...11
C0  ...1.........1...............1......1....
```

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.
- Initialize cursors C_0 using the '&#' stream shifted.

```
T   );8#&( ;54301#& 3443 deed :01#&;3100#& 21
D   ..1.....11111...1111.......11...1111...11
C0  ..1.........1...............1......1.....
```

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.
- Initialize cursors C_0 using the '&#' stream shifted.
- Parallel scan of numeric references with bitstream addition.

T	);8#&(;54301#& 3443 deed :01#&;3100#& 21
D	..1.....11111...1111.......11...1111...11
C_0	..1.........1.................1......1.....
$C_1 = C_0 + D$	.10....100000...1111......100..10000...11

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.
- Initialize cursors C_0 using the '&#' stream shifted.
- Parallel scan of numeric references with bitstream addition.
- Mask off the garbage.

T	);8#&(;54301#& 3443 deed :01#&;3100#& 21
D	..1.....11111...1111.......11...1111...11
C_0	..1.........1.................1......1.....
$C_1 = C_0 + D$	.10....100000...1111......100..10000...11
$C_2 = C_1 \wedge \neg D$	.10....100000.............100..10000.....

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.
- Initialize cursors C_0 using the '&#' stream shifted.
- Parallel scan of numeric references with bitstream addition.
- Mask off the garbage.
- Call out the extracted numeric references.

T	);8#&(;54301#& 3443 deed :01#&;3100#& 21
D	..1.....11111...1111.......11...1111...11
C_0	..1.........1.................1......1.....
$C_1 = C_0 + D$	.10....100000...1111......100..10000...11
$C_2 = C_1 \wedge \neg D$	.10....100000.............100..10000.....
$R = C_2 - C_0$	..1.....11111.............11...1111.....

Parallel Scanning Basics

- Given a text with numeric references, T (little-endian)
- Compute the digits character class stream, D.
- Initialize cursors C_0 using the '&#' stream shifted.
- Parallel scan of numeric references with bitstream addition.
- Mask off the garbage.
- Call out the extracted numeric references.
- Find unterminated reference errors.

T	);8#&(;54301#& 3443 deed :01#&;3100#& 21
D	..1.....11111...1111.......11...1111...11
C_0	..1.........1.................1......1.....
$C_1 = C_0 + D$	.10....100000...1111......100..10000...11
$C_2 = C_1 \wedge \neg D$	.10....100000.............100..10000.....
$R = C_2 - C_0$	..1.....11111.............11...1111.....
$E = C_2 \wedge \neg[;]$	..........................1..................

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.
- A loop sequentially handles attribute/value pairs within tags.

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.
- A loop sequentially handles attribute/value pairs within tags.
- But all tags are processed in parallel!

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.
- A loop sequentially handles attribute/value pairs within tags.
- But all tags are processed in parallel!
- Max iteration count is max # of attributes in any one tag.

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.
- A loop sequentially handles attribute/value pairs within tags.
- But all tags are processed in parallel!
- Max iteration count is max # of attributes in any one tag.
- Block-by-block processing: iterate to max atts for block.

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.
- A loop sequentially handles attribute/value pairs within tags.
- But all tags are processed in parallel!
- Max iteration count is max # of attributes in any one tag.
- Block-by-block processing: iterate to max atts for block.
- Bit stream logic checks for all tag syntax errors in parallel.

Parallel Parsing of XML Tags

XML Start Tag Syntax

```
'<' Name (S Name S? '=' S?
    (('"' [^<"]* '"') | ("'" [^<']* "'")))* S? '>'
```

- Initialize cursors with with [<] bitstream.
- Apply bitstream addition techniques.
- A loop sequentially handles attribute/value pairs within tags.
- But all tags are processed in parallel!
- Max iteration count is max # of attributes in any one tag.
- Block-by-block processing: iterate to max atts for block.
- Bit stream logic checks for all tag syntax errors in parallel.
- Logic is fully implemented within Parabix 2 prototype.

Parsing Results: Call-Out Streams

Document `<top x='1'><a><leaf>4</leaf>;<void/></a></top>`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1.............1.............`
NameFollows	`....1.......1.....1.............1..........`
Names	`.111.......1..1111..........1111..........`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1............1.............`
NameFollows	`....1.......1....1............1..........`
Names	`.111.......1..1111.........1111..........`
AttNames	`.....1.....................................`
AttVals	`.......111.................................`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1............1.............`
NameFollows	`....1.......1....1............1..........`
Names	`.111.......1..1111.........1111..........`
AttNames	`.....1.....................................`
AttVals	`.......111.................................`
Tags	`.111111111..1..1111.........11111..........`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1............1.............`
NameFollows	`....1.......1....1............1..........`
Names	`.111.......1..1111.........1111..........`
AttNames	`.....1.....................................`
AttVals	`.......111.................................`
Tags	`.111111111..1..1111.........11111..........`
EmptyTagMarks:	`.................................1.........`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1............1.............`
NameFollows	`....1.......1....1............1..........`
Names	`.111.......1..1111.........1111..........`
AttNames	`.....1.....................................`
AttVals	`.......111.................................`
Tags	`.111111111..1..1111.........11111..........`
EmptyTagMarks:	`.....................................1.........`
EndTags	`.....................11111.........11..1111.`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1............1.............`
NameFollows	`....1.......1....1............1..........`
Names	`.111.......1..1111.........1111..........`
AttNames	`.....1.....................................`
AttVals	`.......111.................................`
Tags	`.111111111..1..1111.........11111..........`
EmptyTagMarks:	`.....................................1.........`
EndTags	`.....................11111.........11..1111.`
ParseError	`.....................................`

Parsing Results: Call-Out Streams

Document	`<top x='1'><a><leaf>4</leaf>;<void/></a></top>`
NamePosns	`.1.........1..1............1.............`
NameFollows	`....1.......1....1............1..........`
Names	`.111.......1..1111.........1111..........`
AttNames	`.....1.....................................`
AttVals	`.......111.................................`
Tags	`.111111111..1..1111.........11111..........`
EmptyTagMarks:	`.....................................1.........`
EndTags	`.....................11111.........11..1111.`
ParseError	`.....................................`

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.
- The Java Native Interface (JNI) provides access to C, but:

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.
- The Java Native Interface (JNI) provides access to C, but:
  - Java and C/C++ data objects may be incompatible.

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.
- The Java Native Interface (JNI) provides access to C, but:
  - Java and C/C++ data objects may be incompatible.
  - Even string representations are incompatible.

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.
- The Java Native Interface (JNI) provides access to C, but:
  - Java and C/C++ data objects may be incompatible.
  - Even string representations are incompatible.
  - JNI calls are expensive, hundreds of CPU cycles.

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.
- The Java Native Interface (JNI) provides access to C, but:
  - Java and C/C++ data objects may be incompatible.
  - Even string representations are incompatible.
  - JNI calls are expensive, hundreds of CPU cycles.
- But Java is an important and popular technology for XML processing.

The Java Performance Challenge

- Java has no facilities for direct use of SIMD.
- Java's built-in UTF-8 to UTF-16 transcoding is slow.
- The Java Native Interface (JNI) provides access to C, but:
  - Java and C/C++ data objects may be incompatible.
  - Even string representations are incompatible.
  - JNI calls are expensive, hundreds of CPU cycles.
- But Java is an important and popular technology for XML processing.
- High performance solutions must be found.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.
- Array set models may also have other benefits.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.
- Array set models may also have other benefits.
  - Hardware/software prefetching.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.
- Array set models may also have other benefits.
  - Hardware/software prefetching.
  - DMA (direct memory access) hardware.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.
- Array set models may also have other benefits.
  - Hardware/software prefetching.
  - DMA (direct memory access) hardware.
  - SIMD: Use SoA (structure of array) representations.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.
- Array set models may also have other benefits.
  - Hardware/software prefetching.
  - DMA (direct memory access) hardware.
  - SIMD: Use SoA (structure of array) representations.
  - Multicore processors: array models support data partitioning.

Array Set Models

- JNI allows bulk import of arrays of simple types (bytes, integers).
- Therefore model XML data using sets of such arrays.
- Transport array data across JNI boundary in bulk.
- Array set models may also have other benefits.
  - Hardware/software prefetching.
  - DMA (direct memory access) hardware.
  - SIMD: Use SoA (structure of array) representations.
  - Multicore processors: array models support data partitioning.
  - Streaming buffers for large XML documents.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.
  - Node kind, name code, depth and next sibling arrays.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.
  - Node kind, name code, depth and next sibling arrays.
  - Other arrays dependent on type.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.
  - Node kind, name code, depth and next sibling arrays.
  - Other arrays dependent on type.
- NamePools are collections of namespace prefix, URI, local name triples.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.
  - Node kind, name code, depth and next sibling arrays.
  - Other arrays dependent on type.
- NamePools are collections of namespace prefix, URI, local name triples.
- Reimplementation using Parabix/JNI.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.
  - Node kind, name code, depth and next sibling arrays.
  - Other arrays dependent on type.
- NamePools are collections of namespace prefix, URI, local name triples.
- Reimplementation using Parabix/JNI.
  - Copying re-implementation: 2X faster build time.

Initial Study: Saxon-B Data Structures

- An initial study of the ASM concept was to reimplement Saxon-B data structures.
- Two basic structures: TinyTree and NamePool.
- TinyTree uses several arrays of integers.
  - Node kind, name code, depth and next sibling arrays.
  - Other arrays dependent on type.
- NamePools are collections of namespace prefix, URI, local name triples.
- Reimplementation using Parabix/JNI.
  - Copying re-implementation: 2X faster build time.
  - Using direct memory byte buffers: 2.5X improvement.

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.
  - register allocation

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.
  - register allocation
  - rematerialization

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.
  - register allocation
  - rematerialization
  - instruction scheduling

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.
  - register allocation
  - rematerialization
  - instruction scheduling
- Retarget for various instruction set architectures.

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.
  - register allocation
  - rematerialization
  - instruction scheduling
- Retarget for various instruction set architectures.
- Retarget for different memory models.

Automated Generation of Bit Stream Code

- Eliminate tedious, error prone manual coding.
- Apply code generation algorithms.
  - register allocation
  - rematerialization
  - instruction scheduling
- Retarget for various instruction set architectures.
- Retarget for different memory models.
- Adapt for multicore.

Character Class Compiler

- Use character class definitions

Character Class Compiler

- Use character class definitions
  - individual characters

Character Class Compiler

- Use character class definitions
  - individual characters
    - compile(`[CharDef(LAngle, "<")]`)

Character Class Compiler

- Use character class definitions
  - individual characters
    - compile(`[CharDef(LAngle, "<")]`)
  - character ranges

Character Class Compiler

- Use character class definitions
  - individual characters
    - compile([CharDef(LAngle, "<")])
  - character ranges
    - compile([CharSet('Control', ['\x00-\x1F']), CharSet('Digit', ['0-9'])]])

Character Class Compiler

- Use character class definitions
  - individual characters
    - compile([CharDef(LAngle, "<")])
  - character ranges
    - compile([CharSet('Control', ['\x00-\x1F']), CharSet('Digit', ['0-9'])]])
- Generates bit stream code automatically.

Character Class Compiler

- Use character class definitions
  - individual characters
    - compile([CharDef(LAngle, "<")])
  - character ranges
    - compile([CharSet('Control', ['\x00-\x1F']), CharSet('Digit', ['0-9'])]])
- Generates bit stream code automatically.
- Applies various optimizations.

Character Class Compiler

- Use character class definitions
  - individual characters
    - compile(`[CharDef(LAngle, "<")]`)
  - character ranges
    - compile(`[CharSet('Control', ['\x00-\x1F']), CharSet('Digit', ['0-9'])]`])
- Generates bit stream code automatically.
- Applies various optimizations.
- Used in first-generation Parabix.

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- 7 bitwise logical operations required.

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- 7 bitwise logical operations required.
- compile(`[CharDef(LAngle, "<")]`)

Character Class Formation

- Combining 8 bits of a code unit gives a character class stream.
- 7 bitwise logical operations required.
- 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);
  ```
  ```

Character Class Common Subexpressions

- Different characters have common subexpressions.

Character Class Common Subexpressions

- Different characters have common subexpressions.
- compile([CharDef(LAngle, "<"), CharDef(RAngle, "<")])

Character Class Common Subexpressions

- Different characters have common subexpressions.
- 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);
  ```
  ```

Character Class Common Subexpressions

- Different characters have common subexpressions.
- 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 Class Ranges

- Ranges of characters are often very simple to compute.

Character Class Ranges

- Ranges of characters are often very simple to compute.
- compile([CharSet('Control', ['\x00-\x1F']),
  ])])

Character Class Ranges

- Ranges of characters are often very simple to compute.
- 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 Class Ranges

- Ranges of characters are often very simple to compute.
- 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 Class Ranges

- Ranges of characters are often very simple to compute.
- 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);
  ```
  ```

Regular Expression Compilation

- Extend character class compiler for regexp operations.

Regular Expression Compilation

- Extend character class compiler for regexp operations.
- Use bitstream addition for character class repetitions.

Regular Expression Compilation

- Extend character class compiler for regexp operations.
- Use bitstream addition for character class repetitions.
- Ex. [0-9]*

Regular Expression Compilation

- Extend character class compiler for regexp operations.
- Use bitstream addition for character class repetitions.
- Ex. [0-9]*
- Masking for upper or lower bounds: [0-9]{2,5}

Regular Expression Compilation

- Extend character class compiler for regexp operations.
- Use bitstream addition for character class repetitions.
- Ex. [0-9]*
- Masking for upper or lower bounds: [0-9]{2,5}
- Restrict to a deterministic RE subset.

Regular Expression Compilation

- Extend character class compiler for regexp operations.
- Use bitstream addition for character class repetitions.
- Ex. [0-9]*
- Masking for upper or lower bounds: [0-9]{2,5}
- Restrict to a deterministic RE subset.
- Under development.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.
- But, can this be automated?

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.
- But, can this be automated?
- Yes!

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.
- But, can this be automated?
- Yes!
  - Define restricted form of Python bitsream operations.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.
- But, can this be automated?
- Yes!
  - Define restricted form of Python bitsream operations.
  - Translate to SIMD operations using C/C++ intrinsics.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.
- But, can this be automated?
- Yes!
  - Define restricted form of Python bitsream operations.
  - Translate to SIMD operations using C/C++ intrinsics.
  - Compiler handles segmentation into blocks or segments.

Unbounded Bit Stream Compilation

- Recent prototypes use unbounded bitstreams in Python.
- Ex: catalog of XML bit streams.
- Manual coding then used to reimplement in C++.
- But, can this be automated?
- Yes!
  - Define restricted form of Python bitsream operations.
  - Translate to SIMD operations using C/C++ intrinsics.
  - Compiler handles segmentation into blocks or segments.
  - Work in progress.

Concluding Remarks

- Parallel bit stream methods accelerate

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.
  - Whitespace and line break handling.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.
  - Whitespace and line break handling.
  - Sequential parsing using bit scans over lexical item streams.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.
  - Whitespace and line break handling.
  - Sequential parsing using bit scans over lexical item streams.
  - Parallel tag parsing using bitstream addition.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.
  - Whitespace and line break handling.
  - Sequential parsing using bit scans over lexical item streams.
  - Parallel tag parsing using bitstream addition.
- Methods may be used to support any XML API.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.
  - Whitespace and line break handling.
  - Sequential parsing using bit scans over lexical item streams.
  - Parallel tag parsing using bitstream addition.
- Methods may be used to support any XML API.
- Results can be delivered to Java through Array Set Models.

Concluding Remarks

- Parallel bit stream methods accelerate
  - UTF-8 and XML character validation.
  - Transcoding.
  - Whitespace and line break handling.
  - Sequential parsing using bit scans over lexical item streams.
  - Parallel tag parsing using bitstream addition.
- Methods may be used to support any XML API.
- Results can be delivered to Java through Array Set Models.
- Compiler technology promises to ease the programming burden and support retargetting for various architectures.

Knuth's Final Exercise on Bitwise Techniques

217. *[40]* Explore the processing of long strings of text by packing them in a "transposed" or "sliced" manner: Represent 64 consecutive characters as a sequence of eight octabytes $w_0 \ldots w_7$ where w_k contains all 64 of their kth bits.

- Donald E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 1, Addison-Wesley, Boston MA, 2009.

Knuth's Answer

217. See R. D. Cameron, *U.S. Patent 7400271* (15 July 2008); *Proc. ACM Symp. Principles and Practice of Parallel Programming* **12** (2008), 91–98.