HPar: A Practical Parallel Parser for HTML–Taming HTML Complexities for Parallel Parsing

ZHIJIA ZHAO, College of William and Mary
MICHAEL BEBENITA and DAVE HERMAN, Mozilla Corporation
JIANHUA SUN and XIPENG SHEN, College of William and Mary

Parallelizing HTML parsing is challenging due to the complexities of HTML documents and the inherent dependencies in its parsing algorithm. As a result, despite numerous studies in parallel parsing, HTML parsing remains sequential today. It forms one of the final barriers for fully parallelizing browser operations to minimize the browser's response time—an important variable for user experiences, especially on portable devices. This article provides a comprehensive analysis on the special complexities of parallel HTML parsing and presents a systematic exploration in overcoming those difficulties through specially designed speculative parallelizations. This work develops, to the best of our knowledge, the first pipelining and data-level parallel HTML parsers. The data-level parallel parser, named *HPar*, achieves up to 2.4× speedup on quadcore devices. This work demonstrates the feasibility of efficient, parallel HTML parsing for the first time and offers a set of novel insights for parallel HTML parsing

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: HTML parsing, multicore, parallelization

ACM Reference Format:
Zhao, Z., Bebenita, M., Herman, D., Sun, J., and Shen, X. 2013. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. ACM Trans. Architec. Code Optim. 10, 4, Article 44 (December 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555301

1. INTRODUCTION

Research has long shown that reducing application response time is one of the most important variables for user satisfaction on handheld devices [Hoxmeier and Dicesare 2000; Nah 2004]. Multiple studies by industry have echoed the finding in the context of Web browsers: Google and Microsoft reported that a 200ms increase in page load latency resulted in "strong negative impacts" and that delays of under 500ms "impact business metrics" [Mai et al. 2012; Schurman and Brutlag 2009; Kohavi and Longbotham 2007]. One approach to reducing response time that has drawn lots of recent attention is parallelization of Web browser activities, which range from the creation of Web page layout [Badea et al. 2010; Meyerovich and Bodik 2010], lexing [Jones et al. 2009], and Javascript translation and execution [Intel Corporation n.d.] to Web page loading [Wang et al. 2012], XML parsing [Lu et al. 2006; Wu et al. 2008], and Web page decomposing [Mai et al. 2012]. Industry (e.g., Qualcomm and Mozilla) has began

Author's addresses: Z. Zhao, J. Sun, and X. Shen, Computer Science Department, College of William and Mary, Williamsburg, VA, USA 23185; M. Bebenita and D. Herman, Mozilla Corporation, Mountain View, CA, USA 94043.

DOI: http://dx.doi.org/10.1145/2555289.2555301

Table I. Web Page Loading Time and Percentages of HTML Parse
The right two columns show the percentages when the nonparsing parts of the loading are sped up by a factor of two or four through parallelization.

		HTML parsing portion		
		Speed on non-parsing		
benchmark	page loading times (ms)	1X	2X	4X
youtube	2357	8.5%	15.7%	27.1%
bbc	3081	5.6%	10.6%	19.2%
linkedin	7950	16.4%	28.2%	44.0%
yahoo	3962	6.3%	11.9%	21.2%
amazon	2476	19.8%	33.1%	49.7%
qq	6198	10.1%	18.3%	31.0%
twitter	3520	11.9%	21.3%	35.1%
taobao	10891	14.1%	24.7%	39.6%
wikipedia	4771	10.4%	18.8%	31.7%
facebook	4209	17.1%	29.2%	45.2%
geomean	**4397**	**11.1%**	**19.9%**	**32.9%**

designing new browser architectures [Mozilla Corporation n.d.; Cascaval et al. 2013] to leverage these results and to exploit task-level parallelism.

However, despite all of these efforts, an important part of Web browsers—HTML parsing—remains sequential. The second column in Table I reports the loading time of a number of popular Web pages on Firefox 18 on a Nexus 7 tablet. The third column shows the portion taken by HTML parsing, measured through a profiler named *Speed Tracer* (on Chrome 25 on a Macbook Pro, as no such tool was found on Android 4.2 for Nexus 7). On average, the parsing takes 11.1% of the total loading time. According to Amdahl's law, any sequential part will become more critical as the other parts become parallel. As the rightmost two columns in Table I show, when the other activities are sped up by two or four times, the portion of the sequential parsing would increase to 20% or 33%, respectively. Although it is possible that parsing speed may increase with new innovations in parsing algorithms and HTML design, being sequential inherently limits its enhancement and scalability on modern parallel machines. As Web pages become larger and more complex, the issue is becoming more serious.

The goal of this work is to remove this arguably final major obstacle for realizing a fully parallel Web browser.

HTML parallel parsing is an open problem. The difficulties come from the special complexities in the HTML definition. In this article, we focus on the latest definition, HTML5. Unlike most other languages, HTML5 is not defined by clean formal grammars, but by some specifications with many ad hoc rules. An HTML document often contains scripts written in other languages (e.g., Javascript), whose executions during the parsing of the HTML document could add new content into the document. Furthermore, the official HTML organizations have imposed some detailed specifications on HTML5 parsing algorithms. As Section 2 will show, these complex specifications introduce some inherent dependencies into the various parsing operations, making parallel parsing even more difficult. As a result of these special complexities, no parallel parsing techniques have been successfully applied to HTML parsing.

This article presents the first effort to remove the obstacles for parallel HTML parsing. It describes a set of novel techniques and parsing algorithms for taming all of those complexities. Specifically, this article makes the following major contributions:

(1) It presents the first systematic analysis that uncovers the special complexities in parallel HTML parsing.

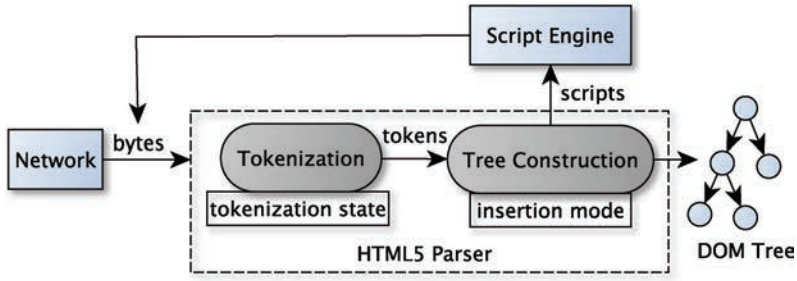


Fig. 1. HTML5 parsing model.

(2) It proposes a set of novel solutions to enable speculative parsing to address the complexities and to circumvent various data dependencies in the parsing.
(3) It constructs the first two parallel HTML5 parsers, one using pipelining parallelism and the other using data-level parallelism.
(4) It evaluates the parsers on a set of real-world workloads. The results show significant speedups (up to 2.4×, 1.73× on average), demonstrating the effectiveness of the (data-level) parallel parser.
(5) It reveals a set of novel insights in constructing parallel HTML parsers:
   (a) Despite being inherently sequential, with a systematic treatment and carefully designed speculation, HTML parsing is possible to run efficiently in parallel.
   (b) It is difficult for pipelined parsing to generate large speedups for HTML parsing.
   (c) The scalability of data-level parallel parsing tends to grow as the Web page size increases. As Web pages continue to get larger [lar], more speedups are expected on future Web pages.

In the rest of this article, we first introduce HTML5 parsing and explain its special complexities. Then, we explore pipeline and data-level parallelism, respectively, and present a set of novel techniques. In the evaluation section, we present and discuss the empirical results on real-world Web pages on different platforms. After a review of related work, we conclude this article with a brief summary.

2. CHALLENGES OF PARALLEL HTML5 PARSING

In this section, we first provide some background on HTML5 and then describe five classes of special challenges it imposes on parallel parsing.

2.1. Background on HTML5 and Its Model of Parsing

HTML5 is the latest version of the HTML standard, with enhanced support for multimedia and complex Web applications. Different from all previous versions, the Web Hypertext Application Technology Working Group (WHATWG) and World Wide Web Consortium (W3C) provide some concrete specifications on the model of HTML5 parsing [htm a] (hereafter referred as HTML5 spec). Previously, the HTML parsers developed by different companies interpreted HTML documents differently, causing inconsistent behaviors among browsers. For instance, IE and Opera read `<foo<bar>` as one tag `foo<bar`, whereas Firefox and Chrome read it as two tags, `foo` and `bar`. For that reason, the HTML5 spec includes specifications on the parsing. According to HTML5 tests [htm b], major modern Web browsers, such as Chrome 23, Safari 5, and Firefox 17, fully support HTML5 parsing rules.

Figure 1 shows the standard HTML5 parsing model [htm a], which contains two stages: tokenization and tree construction. An input byte stream that comes from network first flows into the tokenization stage, where a *tokenizer* parses it into tokens. Upon the recognition of a token by the tokenizer, a *tree builder* immediately consumes

```
<!DOCTYPE html>
<html>
<style type="text/css">
i {color:red;}
li {color:blue;}
</style>
<body>
  <b><i> My List </b></i>
  <ol>
    <li> random picture
    </br>
    <script type="text/javascript">
    var time = new Date();
    if(time.getSeconds()<30)
        var image = "a.jpg";
    else
        var image = "b.jpg";
    document.write("<img src="+image+">");
    </script>
    <li> a table
    <table border="1">
      <thead>
        <tr>
          <th>Month</th>
          <th>Savings</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>January</dt>
          <td><i>$100</i></td>
        <!--<tr>
          <td>February</td>
          <td><i>$80</i></td>
        </tr>-->
      </tbody>
    </table>
  </ol>
</body>
</html>
```

Fig. 2. An example of HTML5.

it and organizes the parsing results in a Document Object Model (DOM) tree. The HTML5 spec uses finite state machines to track the progress of both the tokenization and the tree construction. The states for the tokenization are called the *tokenization state*, and the states for the tree construction are called the *insertion modes*. After parsing, the DOM tree will be passed to the layout engine for Web page rendering. When the tree builder finds some executable scripts (e.g., Javascript code) embedded in the HTML document, the scripts will be executed immediately. The execution may generate some new HTML strings, which also need to be parsed by the HTML parser. Hence, the path from the Script Engine to the input stream.

2.2. Special Challenges for Parallel Parsing

The standard HTML5 parsing model [htm a] designed by W3C and WHATWG consists of complex state machines. The state machine for tokenization, for instance, contain 68 states, and more than 250 transitions among those states. The complex parsing model just reflects the tip of the iceberg for the development of a parallel HTML parser. There are more inherent complexities preventing the HTML parser running in parallel. In this part, we use the example in Figure 2 to explain these complexities.

Informal Language. Unlike many other languages, HTML5 is not defined with some clean, formal grammars,[1] but a collection of ad hoc rules and specifications. Most

[1]We are aware of some Antlr grammars for earlier HTML versions, but they do not conform to HTML5 spec.

previous work on parallel parsers [Alblas et al. 1994; Baccelli and Fleury 1982; Baccelli and Mussi 1986; Luttighuis 1989; Fischer 1975; Skillicorn and Barnard 1989] has targeted formally defined languages (e.g., in a context-free grammar) and leveraged the features of the grammars for parallelization. The ad hoc definition of HTML5 makes these techniques hard to be applied to its parsing.

Error Correcting. As many existing HTML documents contain syntax errors, Web browsers have been tolerating such errors with some kind of automatic error correction. Now, this feature is formally integrated into the official HTML5 parsing model. As a result, HTML5 parsers typically resolve some common errors to their best knowledge. When validating it with W3C's HTML5 conformance checker, the example in Figure 2 contains three syntax errors: missing `title` element, bad nested tags between `<b>` and `<i>` in line 8, and a tag name typo `</dt>` in line 30. Unlike the conformance checker, HTML5 parsers typically do not report errors to end users, but instead try to fix them. For instance, they would ignore the typo tag `</dt>` since no start tag `<dt>` was met before. When the next `<td>` is read, they would assume a `</td>` is missing and automatically insert it into the token flow. This forgiveness is essential for working with real-world HTML documents but complicates parallel parsing. For instance, suppose the HTML document in Figure 2 is split into two parts and assigned to two threads to parse in parallel. When the second thread encounters `</dt>`, it does not know whether this unmatched tag is an error or has its partner tag `<dt>` in the first part of the document that this thread does not see. The parallel parsing algorithm has to correctly handle such ambiguities.

Embedded Languages. It is common for an HTML5 document to contain some embedded code written in other scripting languages (e.g., CSS, Javascript, MathML, and SVG.) Theoretically, there is no restriction on what language can be used, as long as the embedded codes are marked by the `<script>` tag. In Figure 2, a CSS segment is embedded at lines 4 and 5, whereas a Javascript segment is between lines 13 and 18. These embedded languages bring their own syntaxes into HTML5 documents, complicating the parsing. Previous work on parallel parsing has not considered such complexities.

Dynamic Inputs. Another complexity coming with the embedded scripts is that they may alter the input HTML5 document. As shown in line 18 of Figure 2, the script `document.write()` inserts extra tags into the HTML5 input stream. The inserted tags may even introduce new errors to the HTML document, which will not be detected before the script is executed. In our example, an `alt` attribute in the tag `<img>` is required by HTML5 spec but is missing there. This feature introduces hazards or dynamic dependencies into the parallel parsing of HTML documents, because the changes the scripts make to the input document (e.g., generating a new tag) may affect how the later part of the document should be parsed.

Strong Dependencies. As the next section will show, due to the unique features of HTML, cyclic dependencies exist between tokenization and tree construction, as well as among the processing of different parts of the HTML document, making neither pipelining nor data-level parallelization directly applicable.

Because of these five special complexities, previous techniques developed for parallel parsing other languages have not been applied to HTML successfully. In this work, we address these complexities through a carefully designed speculative parallelization, which exploits the special features of HTML, as well as some insights drawn from the statistical properties of real-world HTML files. Our explorations cover both pipelining and data-level parallelism. We next present each of them.

3. SPECULATIVE PIPELINING

Pipelining is a parallelization technique in which the execution process is divided into stages, and the data go through the stages one by one so that the stages can run in parallel. This technique has been used in various layers of computer systems for performance. In this section, we present a lightweight pipelining algorithm for parallel HTML5 parsing.

To form a pipeline, one must first decouple the original task into multiple stages. A data element goes through all of the stages. The processing of a data element in a stage depends on the earlier stages, but different stages may process different data elements concurrently. So a pipeline has the potential to produce a speedup up to the number of stages. But the actual speedup is often lower than that due to the workload distribution among stages and synchronizations between threads [Vachharajani et al. 2007].

The HTML5 spec suggests that HTML parsing can be naturally decoupled into two stages: tokenization and tree construction. Our manual analysis of some typical HTML5 parsers (such as jsoup [jso] and Validator.nu [val]) shows that these two stages have the least number of dependent data structures between them and have a clear interaction interface. Finer-grained ways to partition the parsing are possible. But the complex interactions would result in more complicated code and tremendous difficulty for later maintenance and extension. We hence focus our implementation on this two-stage pipeline.

Even for this natural two-stage partition, directly applying the basic pipelining scheme to them cannot expose any parallelism. We next analyze two special challenges for pipelining HTML5 parsing and then explain how our speculative pipelining solves those problems.

3.1. Pipelining Hazards

Data dependencies that prevent a pipeline stage from processing the next item before its following pipeline stages finish processing the current item are called *pipelining hazards*. In the HTML5 parsing model, there are a series of such hazards that restrain the tokenizer and tree builder from running in parallel.

Tokenization State. The HTML5 spec requires that the tokenizer cannot start processing new data elements before the tree builder finishes consuming the newly recognized token. The reason for this requirement is that the tree builder may modify the tokenization state during its consumption of the newly recognized token. If the tokenizer continues to generate the following tokens regardless of the tree builder's status, it may start from a wrong tokenization state, leading to tokenization errors.

Figure 3 shows such an instance happening when the example in Figure 2 is parsed. In a sequential parsing, when the tree builder sees the `start style` tag, it expects that the following token must be from CSS code and hence correctly switches the tokenization state from `DATA` to `Rawtext`. However, in a pipelined parsing, if the tokenizer does not wait for the tree builder, it would tokenize the following input elements before the tokenization state gets updated and hence produce the wrong tokenization results, as Figure 3 shows.

Based on the HTML5 spec, we identify 10 cases, listed in Table II, when the tree builder may change the tokenization state during its processing of tokens. The HTML5 spec puts the responsibility of updating the tokenization state on the tree builder because the new state depends on both the new token and the insertion mode of the tree builder. For instance, as the first row of Table II shows, when the tree builder receives a token `<title>`, it would switch the tokenization state to `RCDATA` only when the insertion mode of the tree builder is `InHead` (which means that the parser is parsing the region in

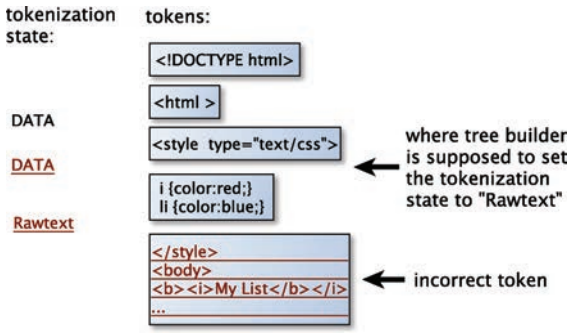


Fig. 3. An example showing that the naive pipelined HTML5 parser emits incorrect tokens, marked with underlines.

Table II. Pipelining Hazards from Tokenization State

Insertion Mode	Next Token	Switch-to State
InHead	<title>	RCDATA
InHead	<noframes>	RAWTEXT
InHead	<style>	RAWTEXT
InHead	<script>	script data
InBody	<xmp>	RAWTEXT
InBody	<iframe>	RAWTEXT
InBody	<noembed>	RAWTEXT
InBody	<noscript>	RAWTEXT
InBody	<plaintext>	PLAINTEXT
InBody	<textarea>	RCDATA

the scope of `head`).[2] The tokenization state hence causes cyclic dependencies between the tokenizer and tree builder.

Self-Closing Acknowledged Flag. A more obscure pipelining hazard in HTML5 spec is the *self-closing acknowledged flag*, a global variable in the tokenizer. In HTML, a set of tags are predefined to be self-closing tags, meaning that they need no separate end tag. The new line tag `<br/>` is such an example. A tag with "/" at the end is considered as being written in a self-closing form; a predefined self-closing tag should be written in such a form but does not have to be—another evidence of the flexibility of HTML5 spec.

The self-closing acknowledged flag is used in the parser to assist the check on whether a tag written in a self-closing form (i.e., with "/" as its ending symbol) indeed belongs to the predefined set of self-closing tags. According to the HTML5 spec, the tokenizer and tree builder must cooperate to do the check. The global variable self-closing acknowledged flag is set to true initially. When the tokenizer recognizes a token, if the ending symbol of it is "/", it pessimistically sets the flag to false; otherwise, it keeps the self-closing acknowledged flag unchanged. Then, when the tree builder receives this start tag, it will check if this tag is in the predefined self-closing tag set. If so, it sets the self-closing acknowledged flag to true. Next, right before the tokenizer attempts to process the next symbol in the input, it checks the self-closing acknowledged flag and records an error if that flag is false, indicating that the token just processed is not a self-closing tag but is mistakenly written in a self-closing form in the input. So, in a naive parallel processing, if the tokenizer continues to generate the next token before

[2]The insertion mode and next token are not the only conditions that cause the tokenization state to switch; *script flag*, for instance, can also trigger it.

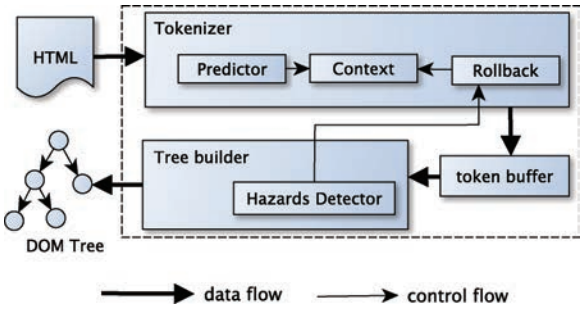


Fig. 4. The speculative pipelining framework for HTML5 parsing.

the tree builder finishes processing the current token, it would miss the self-closing error that the tree builder would find when processing the current token, leading to wrong parsing results on the following tokens.

The cyclic dependencies between the tokenization and tree construction serialize the two stages. As a result, a basic pipelining scheme can exploit no parallelism in the HTML parsing.

3.2. Speculative Pipelining

To overcome the challenges, an attempting option is to remove all of the dependencies by redesigning the state machines in the tokenizer and tree builder. But the cyclic dependencies between them are inherent rather than side products of the state machines. For instance, one may let the tokenizer, rather than tree builder, get in charge of the tokenization state update. But that does not address the problem, because in either design, the update depends on the information from both the tokenizer (the next token) and the tree builder (the insertion mode). The same happens to the self-closing acknowledged flag update. Even if some dramatic changes to the parser might go around these issues, they would likely seriously violate the HTML5 spec.

We instead address the challenges by developing a *speculative pipelining* scheme. Speculative pipelining is a pipelining technique that employs speculation to break the cyclic dependencies between stages. Its basic structure contains three steps: (1) predict the value of the data items that cause the dependencies; (2) with the predicted values, process the tasks in pipeline speculatively; (3) when the dependent data is produced, verify the prediction, and if the prediction is wrong, trace back and reprocess the data from the correct state (called *rollback*).

Our design is based on some empirical observations. We conduct a systematic measurement on how often the tree builder alters the tokenization state. Our measurement on the top 1,000 Web sites [top] shows that it happens in less than 0.01% of all operations of the tree builder. It suggests that if the tokenizer optimistically assumes that its state will not be altered by the tree builder, there is less than 0.01% probability that it has to roll back. Moreover, we find that among the 117 HTML5 tags [tag], only 18.8% of them are self-closing, suggesting that speculation may be sufficient to overcome the self-closing acknowledged flag hazard as well.

Based on these observations, we develop a speculative pipelining framework for HTML5 parsing, as shown in Figure 4. This framework contains four main components: a predictor that predicts the context (including the tokenization state and self-closing acknowledged flag) for tokenization, a hazard detector that detects the pipelining hazards, a rollback function that recovers the context from misspeculation, and a token buffer for passing the tokens from the tokenizer to the tree builder. We explain each of the components next.

```
class Snapshot {
    TokenizerState state;
    boolean acknowledgeSelfClosingFlag;
    Token.StartTag lastStartTag;
    ...
}
abstract class Token {
    TokenType type;
    int index;
    Snapshot snapshot; /* new */
    ...
}
abstract class TreeBuilder {
    CharacterReader reader;
    Tokenizer tokenizer;
    protected Document doc;
    ...
    private Token previousToken; /* new */
    ...
}
```

Fig. 5. Data structures for rollback.

Predictor. Basically, there are two ways to predict the context. One is optimistic prediction, in which we expect that no hazards will happen. So the context after emitting the prior token is considered as the current context. The other is pessimistic prediction—that is, we do not expect the prior context to be a good guess but use an advanced predictor (e.g., a predictor that exploits conditional probabilities based on partial contexts [Zhao et al. 2012]) to gain good speculation accuracy. There is a classic trade-off between prediction overhead and prediction accuracy. In this work, we choose the optimistic prediction because of its simplicity and the low probabilities of the hazards. In Section 5, we will show that the design provides sufficient accuracy.

Hazard Detector. To detect pipelining hazards, we use a flag, *hazard flag*, to indicate whether some pipelining hazards have occurred. For example, when the tree builder reads a start tag whose name is "plaintext," it switches tokenization state to `PLAINTEXT`. Right after this, it sets the hazard flag to true. The pipelining hazards mentioned in Section 3.1 scatter over the entire tree construction algorithm. To cover them, we look into the HTML5 spec and identify all sites in the algorithm where a pipelining hazard may happen. For each site, we insert a corresponding detection instruction. In this way, the detector is fully integrated into the tree builder and is automatically triggered by any pipelining hazard.

Rollback. When a pipeline hazard occurs, the tokenizer may run into some incorrect states. To effectively roll back from the incorrect status, two questions need to be answered. The first is to which place should the tokenizer roll back? This place should be safe and should be as late as possible to minimize the rollback overhead. Our solution meets both criteria by moving the tokenizer back to a state where the tree builder just finishes consuming the token that causes the hazard. The second question is how can the status be recovered at the safe point? Our solution is to add a *Snapshot* data structure that encapsulates the data needed for recovering and a *previousToken* variable that contains the needed snapshot, as shown in Figure 5.

Our solution introduces a hazard flag. The hazard detector sets the flag once it finds some hazards. The tokenizer checks the flag every time before it starts to produce the next token; thus, usually no more than one token is produced between the time the flag is set and the time the tokenizer sees it. Once the tokenizer sees that the flag is set, it invokes the rollback and then resets the flag. The overhead of rollback mainly consists of the flag checking and recovery of the tokenization state.

```
<lists>   ::= <list> <lists>
            | <empty>

<list>    ::= <list> <items> </list>

<items>  ::= <item> <items>
            | <empty>

<item>    ::= <item> <data> </item>
            | <list>

<data>  ::= [a-zA-Z ]*
```

```
<list>
  <item>pen</item>
  <list>
    <item>laptop</item>
    <item>tablet</item>
  </list>
  <list>
    <item>cartoon</item>
    <list>
      <item>Jane Eyre</item>
      <item>Oliver Twist</item>
    </list>
  </list>
</list>
```

Fig. 6. The LIST language and its example input.

Token Buffer. The token buffer plays the "pipe" role in this pipelining scheme. Its efficiency is important for the pipelining performance. We initially employ a nonblocking concurrent queue, which implements an efficient "wait-free" algorithm [Michael and Scott 1996].

In our experiments, we empirically found that the tokenizer and tree builder generally have similar rates in terms of token buffer operations (i.e., dequeue and enqueue). It suggests that a small buffer may be sufficient for performance. We tried a spectrum of buffer sizes. As Section 5.2 will show, the smallest buffer size, 1, gives performance similar to other sizes. It is hence adopted in our design. At a pipeline hazard, the tokens in the queue have to be discarded.

4. SPECULATIVE DATA-LEVEL PARALLELIZATION

Data-level parallelization is orthogonal to pipeline parallelization (they could be used together). It partitions the input into a number of chunks and distributes them to processors for parallel processing. Applying it to HTML5 parsing faces the five complexities discussed in Section 2.2. This section presents our solution—a speculative scheme that exploits the special features of HTML and statistical properties of real-world HTML files.

To ease the explanation, we start with a simpler problem: parallelizing the parsing of documents in LIST, a language that we made up with much simpler features than HTML. The discussion offers some important intuitions for parallelizing HTML5 parsers. We then describe how we capitalize on those insights for developing our data-level parallel HTML5 parser.

4.1. Insights from a Simplified Parsing Problem

4.1.1. The Problem of Parallel LIST Parsing. Earlier HTML specifications are based on the Standard Generalized Markup Language (SGML). These versions of HTML can be defined by Document Type Definition (DTD). LIST is an instance of such DTD-defined languages. Figure 6 gives the definition of LIST and one legal input. To explore data-level parallelism, our goal is to design a parallel parsing algorithm with three major functions: *partition*() for partitioning the input into smaller units, *parallelParsing*() for running a number of parsers on the smaller units, and *merge*() for merging parsing results into a complete one.

4.1.2. Algorithm Design Space. At the core of designing a data-level parallel parsing algorithm for LIST are the designs of the partitioning function *partition*() and merging function *merge*(). Based on different emphasis put on the two functions, there are two directions: being either partitioning oriented or merging oriented.

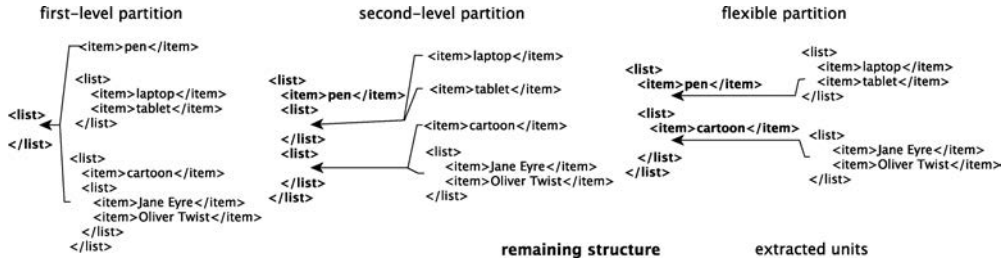


Fig. 7. Partitioning results of the input in Figure 6 through three different partitioning strategies in the partitioning-oriented scheme (bold font for remaining structures and normal font for extracted units).

4.1.2.1 Partitioning-Oriented Scheme. In this scheme, the partitioning function *partition*() is carefully designed to take advantage of the structure of input. Before regular parsing, the *partition*() makes a quick parse of the input, called *preparse*(), to extract the input units from the original input by traversing the input up to a certain number of levels of structure. The extraction result is called *high-level structure* (level zero is the highest). Depending on the number of levels that *preparse*() considers, there are two partitioning strategies: fixed-level partition and flexible partition, as shown in Figure 7.

The fixed-level partition extracts input units from some fixed number of levels of the input structure, whereas the flexible partition extracts an input unit if and only if its size exceeds a given threshold. Which one is better depends on the input structure. A good partition should meet two criteria: first, the extracted input units can be evenly grouped and distributed to a given number of parsers; second, the size of the high-level structure should be small, as the extraction step is sequential.

After *parallelParsing*(), each parser outputs one or more small DOM trees, one for each input unit. The parser that parses the high-level structure produces a special DOM tree, which offers the guide for merging the small DOM trees into a complete one. In this scheme, the *merge*() function is straightforward to implement.

4.1.2.2 Merging-Oriented Scheme. The merging-oriented scheme follows an opposite design principle. In this scheme, the *partition*() is simple, just cutting the input into chunks (almost) evenly. Note that the cut may break some pairs of tags. To handle this special kind of unmatched tags, the parsers record them during *parallelParsing*() and resolve them during the merging stage.

In this work, we choose the merging-oriented scheme over the partitioning-oriented scheme for two reasons:

(1) To take advantage of the structure of input, the partitioning-oriented scheme requires a *preparse*() step, which sequentially scans the entire input. The time complexity is $O(n)$, where n is the number of total tokens.[3] In contrast, the merging-oriented scheme has no such requirement. Even though it has a more complex merging process, the merging cost is only $O(log_2 n)$, as proved later in this section.
(2) Compared to the even cut of input in the merging-oriented scheme, the partitioning-oriented scheme is subject to load imbalance. When the high-level structure becomes the bottleneck, it may further worsen the issue.

4.1.3. Tree Merging–Based Parallel Parsing Algorithm Temppa. In this subsection, we introduce Tree Merging–Based Parallel Parsing Algorithm *Temppa* for LIST. Temppa is a

[3]Parallelizing the *preparse* may help reduce the overhead, but only modestly due to the I/O-bound property of *preparse* [Wu et al. 2008].

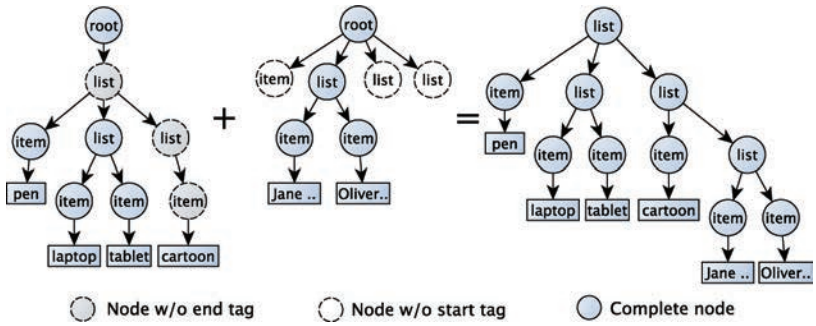


Fig. 8. The DOM trees produced and merged by the *Temppa* algorithm on the input shown in Figure 6. The input is partitioned right after the word "cartoon." Note that *root* is an assistant node and is removed after tree merging.

merging-oriented scheme (see Figure 9, which shows the core of this algorithm). We will use the example in Figure 6 for our explanation.

At first, the input is partitioned into chunks. The partition tries to be even in size and meanwhile avoid breaking tag names. The example input in Figure 6 is partitioned into two chunks at the point right after "cartoon" at line 8. Then, for each chunk, the *parse*() function creates a DOM tree, as shown as the left two trees in Figure 8. Each nonleaf node in a DOM tree represents a construct in the input LIST document. The node hence usually corresponds to the start tag and end tag of the construct in the input document. An example is the leftmost "item" node in Figure 8, whose start and end tags are the first "<item>" and "</item>" in the input shown in Figure 6.

During parsing, the *parse*() function uses a stack to store incomplete tag tokens and pops them off when finding their matching tags. Some nodes in the tree remain incomplete to the end. That happens when the input partition separates a pair of tags into two chunks (or errors in the input). For instance, the "item" node right above "cartoon" in the first tree is an incomplete node without an end tag (called an *end-missing node*), whereas the leftmost "item" node in the second tree is incomplete for missing a start tag (called a *start-missing node*). Other incomplete nodes in the two trees correspond to the lists enclosing the broken item. Note that all tokens with start tag missing are put as immediate children of the root and are always attached as the rightmost node (line 21 in Figure 9); this helps merging efficiency, as discussed next.

The *merge*() function merges every tree into the first. The algorithm is designed by leveraging the following important property of the DOM trees:

PROPERTY 4.1. *Single-path property: for a LIST document with no missing tags, a DOM tree from Temppa can have at most one path that contains end-missing nodes.*

A path in a DOM tree here refers to a branch spanning from the root node to a leaf node. For instance, the branch from "root" to "cartoon" in Figure 8 is a path that contains three end-missing nodes. We call a path with end-missing nodes an*end-missing path*. The single-path property claims that one DOM tree cannot have more than one end-missing path.

This property can be proved easily. Suppose there are two such paths, and $node_i$ and $node_j$ are two different end-missing nodes in them, respectively. Without loss of generality, assume that the start tag of $node_i$ appears before the start tag of $node_j$ in this chunk of the LIST document. There are only three possible cases regarding the end tag of $node_i$: it appears before the start tag of $node_j$, after the end tag of $node_j$, or in between. The first case is impossible in our setting because $node_i$ would be a complete

```
1  // run by each parsing thread
2  Node parse() {
3    stack = new Stack<Node>();
4    current = root;
5    while(true)
6      token = tokenize();
7      switch(token.type)
8        case startTag:
9          //Node(parent, data, isComplete)
10         node = new Node(current,token,false);
11         current.add(node);
12         current = node;
13         stack.push(node);
14         break;
15       case endTag :
16         if(stack.top == token)
17           current.isComplete = true;
18           current = stack.pop();
19         else
20           node = new Node(current,token,false);
21           root.add(node);
22         break;
23       default :
24         node = new Node(current,token,false);
25         current.add(node);
26   return root;
27 }
28
29 // runs on main thread
30 Node merge() {
31   for(i = 1; i < numThreads; i++)
32     root[0] = mergeTwo(root[0], root[i]);
33   return root[0];
34 }
35
36 // merge two incomplete trees
37 Node mergeTwo(Node root0, Node rooti) {
38   endMissingNode =
39           getLowestEndMissingNode(root0);
40   for(child in rooti.children)
41     if(child.StartMissing == true)
42       if(endMissingNode.name == child.name)
43         endMissingNode.EndMissing = false;
44         endMissingNode = endMissingNode.parent;
45       else
46         error("mismatched tags");
47     else
48       endMissingNode.add(child);
49   return root0;
50 }
```

Fig. 9. The core of the *Temppa* algorithm.

token, as both its start and end tags appear in this chunk of document. The second case is not possible either because $node_j$ would be a child of $node_i$ (and hence appears on the same path), as its corresponding construct would be embedded in that of $node_i$. The third case is impossible for a legal LIST document because it would violate the hierarchical structure of LIST documents. (If it does appear in an illegal LIST file, according to the autocorrection rules of HTML, the parser would create an end tag for $node_j$ before the end tag of $node_i$ and ignore the later appearance of the end tag of $node_i$. That would make $node_j$ a child of $node_i$.)

With that property, merging $tree_i$ to $tree_0$ just needs to check the end-missing path in $tree_0$ from bottom to top and change a node's status to complete if its end tag is found in $tree_i$. The algorithm is shown in Figure 9. Here we highlight several features designed for efficiency. First, during the check, the algorithm only needs to examine

the immediate children of the root in $tree_i$ (line 40 in Figure 9) because the *parse*() function puts all start-missing nodes at that level. Second, thanks to the way that *parse*() attaches nodes into a DOM tree, the left-to-right order of the start-missing nodes aligns exactly with the bottom-to-top order of the nodes in the end-missing path; both correspond to the inner-to-outer nesting levels of the broken tokens. Two trees are connected at line 48 in Figure 9. In our example, the topmost list node of the second tree is attached as a child of the rightmost list node at the third level of the first tree. After merging, the root node of the tree is removed.

Let n be the number of tokens the input document contains. The height of the complete DOM tree is $O(log_2 n)$, hence the complexity of the merging.

Temppa cannot directly apply to HTML5 due to the special complexities and dependences in HTML5 parsing. We next present how we address these complexities by integrating carefully designed speculation into Temppa.

4.2. HPar: Breaking Data Dependencies with Mixed Speculation

In this part, we describe *HPar* (*H* for "HTML"; *Par* for "Parallel Parsing"), our data-level parallel HTML parser. *HPar* uses the merging-oriented parallel parsing scheme. As known, data dependence is the single important barrier for parallelization. The various types of complexities of HTML5, as listed in Section 2.2, are ultimately reflected by their impact on complicating data dependencies in parsing. Our description will hence focus on how HPar breaks the many special data dependencies through a set of effective speculations; the treatment of the many complexities of HTML5 is described along the way.

As Section 3 has mentioned, the dependencies in HTML5 parsing can be categorized into tokenization dependencies and tree construction dependencies. Although they may interact with each other, we describe our solutions to them separately for clarity.

4.2.1. Tokenization Dependencies. Suppose an HTML file has been partitioned into chunks. To let an HTML5 parser process an HTML chunk, it needs to know the *starting tokenization state*. The real starting tokenization state depends on the parsing results of previous chunks. So before parallel parsing of HTML chunks, only the parser for the first chunk is sure of the real starting state, according to HTML5 spec, the `DATA` state; all other parsers have to speculate their values. To speculate starting tokenization states, we employ a series of techniques and combine them to achieve a high speculation accuracy.

a. Smart Cutting. A simple but efficient way to improve the accuracy is to select a good cutting point. The policy used in this work and some previous papers [Wu et al. 2008] is to always cut just before (or after) a tag. It starts from an even partitioning point and looks forward until it finds the left angle bracket of a tag and uses that point as a cutting point. This strategy guarantees that no tags are broken and limits the prediction targets to these restricted scenarios. As only a subset of tokenization states are feasible in these scenarios, the strategy simplifies speculation. It may cause some load imbalance, but by at most one token; the effect is typically negligible.

b. Frequency-Based Speculation. A straightforward way to get a statistically good candidate is to choose the state with the highest probability. We profiled the top 1,000 popular Web sites [top] and collected the tokenization state distribution. In this distribution, the `DATA` state has the highest probability: 12.3%. In other words, choosing the `DATA` state as the starting tokenization state can achieve a 12.3% accuracy. However, when we apply the *smart cutting* to restrict the scenarios to predict, the `DATA` state appears to be the true state in 92.3% of the cases. So the *smart cutting* and

this frequency-based speculation together offer a simple but effective way to achieve a 92.3% speculation accuracy.

c. Speculation with Partial Context. The previous two techniques explore static characteristics to help speculation. Although they work in most cases, they cannot handle cases when a comment or script is broken by the cutting. Although these two cases do not happen often in the parsing of most documents, they appear to be critical for some files where comments or scripts are used intensively. We use *partial context* to address these issues. The method exploits the conditional probabilities of partial context—which here refers to the prefix of HTML chunks. We explain the two cases as follows:

(1) Handling Broken Comments. Broken comments happen when different parts of a comment are partitioned into different chunks. For example, when an HTML chunk ending with "`<!--<h2>heading`" is being parsed, its following input elements "`</h2>--><h1>`" are being concurrently parsed. The second parse would mistakenly consider the actually commented-out tag "`</h2>`" as a normal tag. The cause of this error is that the second parse starts from the default tokenization state `DATA`, although the real state should be `Comment`. To solve this problem, we introduce two new token types, `StartComment` and `EndComment`, so that the broken comments can be recognized individually and merged later. The construction of the `StartComment` is simply triggered by an encounter of "`<!--`", whereas an `EndComment` token is built when "`-->`" is encountered and this current HTML chunk contains no matching starting comment tag. The parser would include all text from the starting of the chunk to this end comment tag into that `EndComment` token. Note that these two token types are intermediate representations. After merging, they will be replaced by `Comment` nodes.

The technique just shown has risks and benefits, as it solves the broken comments problem but brings some side effects. For example, when parsing an HTML tutorial page, a parser thread would mistakenly consider the normal text segment "`<p>-->`" in text "`<p>--> ends an HTML comment`" as an `EndComment`. To solve this dilemma, we simultaneously maintain two alternative parsing results: normal interpretation and `EndComment` interpretation. Which one is correct depends on the parsing results of previous chunks. Specifically, if the prior chunk has a `StartComment` as its rightmost incomplete node, then `EndComment` interpretation is correct and used; otherwise, the normal interpretation is used. This approach safely avoids rolling back in either case.

(2) Handling Broken Scripts. Another interesting case involves broken scripts. When a parser thread starts in the middle of a pair of `script` tags, it may not realize this until it reads a `</script>` token. During the merging of two incomplete trees, if the first tree has an incomplete `script` node without an end tag, although the first child of the second tree's root is an incomplete `script` node without a start tag, these two nodes will be merged to form a complete `script` node. But this is not the end of the story. Due to the broken `script` tags, the script cannot be executed until it is complete, hence a delay for its execution results to take effect. As the scripts may modify the HTML input and hence the DOM tree, the delay may cause side effects: changing the application scope of the script. As shown in Figure 10, the script in the piece of HTML is supposed to change only the first two `h2` tags; however, since its execution is after parallel HTML parsing, it changes the third `h2` tag as well. The key to solving this issue is in remembering the application scope of scripts by inserting some anchors into the DOM tree. Each anchor indicates the end of the scope for a broken script to take effect. After the execution of a merged complete script, if it alters the HTML document, the parser reparses the affected chunks and remerges them into the tree. All anchors are removed after the execution of all scripts.

```
<h2>head 2.1</h2>
<h2>head 2.2</h2>
<script type="text/javascript">
var el = document.getElementsByTagName('h2');
for(var i=0; i<el.length; i++) {
   el[i].innerHTML='changed head';
}
</script>
<h2>head 2.3</h2>
```

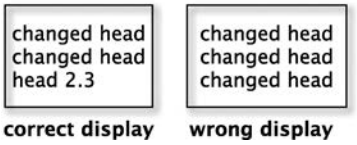


Fig. 10. An example that shows the application scope of script may be changed by the delay of its execution.

(3) Handling Other Broken Tags. There are some other cases that cause the speculation to fail. For example, as shown in Figure 3, when a pair of `style` tags are broken, the `DATA` will not be the true tokenization state. This can be easily fixed by cutting just before a tag, instead of after a tag, assuming the symbol "<" does not appear within `style` tags. Other cases, such as broken `svg` and `math` tags, are handled similarly.

4.2.2. Tree Construction Dependencies. Besides tokenization dependencies, we also need to solve the tree construction dependencies. For a parser thread to process an HTML chunk, it needs to know the *starting insertion mode*. In this subsection, we show that the dependencies can also be addressed through mixed speculations.

a. Before Speculation. Autocorrection is one of the key features in HTML parsing. This feature makes the HTML syntax more flexible and tolerant to some errors, but it makes parallelization difficult. For instance, the Temppa algorithm is supposed to attach the token of an unpaired end tag to the *root* node. However, autocorrection may instead regard the tag as "errors" and let the parser ignore it. We adopt a simple solution that delays this type of autocorrection until the merging stage. The parser always attaches unpaired end tags to the tree and removes them from the tree if the parser finds no matching start tags for them at the tree merging time.

The other type of error handled by autocorrection is missing end tags (e.g., missing "`</li>`" in "`<li> item one. <li> item two.`"). By default, an HTML parser is supposed to autocorrect such errors by inserting some end tags according to some rules in the HTML5 specification. What end tags to insert depends on the current *insertion mode* and the top of an *open stack* that records all start tags not yet matched with end tags. In HPar, the *insertion mode* is determined through speculation as detailed next. As to the open stack, before starting parsing a chunk, a parser determines the top of the open stack by finding the last tag in the suffix of the previous chunk. If it is a start tag, the parser puts it to the top of the open stack; otherwise, it leaves the stack empty. During the parse, it uses the speculated insertion mode and the open stack to autocorrect missing end tags. If the autocorrection hits the bottom of the open stack, the parser conservatively assumes that the open stack does not contain enough information for the autocorrection, aborts, and waits for the previous parser to reparse this chunk. Our study shows that most missing end tags are "`</li>`", which were always fixed by the speculation scheme. No autocorrection-triggered abort is observed in our experiments.

b. Profile-Based Speculation. Similar to the tokenization state, we also profiled the distribution of insertion mode based on the top 1,000 popular Web sites [top]. We

Table III. Insertion Mode Prediction

Table-Related Tokens	Starting Insertion Mode
</td>	InCell
</tr>	InTableText
</th>	InCell
</thead>	InTableText
</tbody>	InTableText
</tfoot>	InTableText
...	...

find that the `InBody` mode is the dominating mode, with a 51.3% probability. However, directly using `InBody` as the starting insertion mode can cause some side effects because of missing `html`, `head`, and `body` elements. For example, when a pair of *head* tags is broken, if the `InBody` mode is used, the parser that processes the second piece would mistakenly consider that all tags it encounters are within the scope of `body` and attach the corresponding elements to `body`. To avoid this, we instead use the `Initial` mode as the starting state for the parsing of each HTML chunk such that the elements `html`, `head`, and `body` will be automatically inserted. And right after those insertions, the insertion mode will automatically switch into the most probable state: the `InBody` mode.

c. Speculation with Partial Context. Partial contexts can also help the speculation of insertion mode. We explain it by drawing on broken `table` tags as an example.

In the HTML5 specification, there are a set of insertion modes designed for parsing table-related tokens, including `InTable`, `InTableText`, `InRow`, and others. This set of insertion modes are important: They cover more than 38% of the insertion modes during the parse of the top 1,000 Web sites.

When a cutting point is within a pair of `table` tags, the speculation with `InBody` as the starting insertion mode will fail. The true insertion mode depends on the position of the cutting point within the pair of `table` tags. So we profiled the relations between the relative positions of cutting point and the insertion modes and found an interesting result: in the situations where `table` tags are broken, the correct starting insertion mode strongly correlates with the first table-related token in the current HTML chunk, as summarized in Table III. We integrate the table into the parallel parser to help the speculation.

For example, suppose a parsing thread starts in the middle of a pair of `table` tags. At first, it does not realize this and starts with the `InBody` mode. After processing a number of tokens, it reads a token </td> without previously reading any start tag <td>. It realizes that earlier speculation fails. Based on Table III, it picks the insertion mode `InCell` as the new speculation of starting insertion mode and reparses its HTML chunk.

Even with all of the techniques for effective speculation, mistakes can still happen in some cases. We treat them in an eager manner: as soon as the parser realizes that the speculation is wrong (by, for instance, encountering an end tag that is inconsistent with the assumed insertion mode), it immediately picks a more plausible speculation state and starts to reparse the chunk of input. The refinement of the speculation state can leverage everything that the parser has observed so far. However, in our exploration, we found that the most useful info is the tag or token the parser encounters right before it realizes the error. For instance, if the tag is <meta>, it would speculate that the beginning of the chunk is more likely to be part of a `head` tag pair (than a `body` tag pair). The refined speculation may still need to be rolled back if it is mistaken. In the worst case, the mistake could not get fixed until the finish of the parsing of all previous chunks of input, when the correct starting states for parsing this chunk

Table IV. Benchmark Size and Speculation Accuracy

web site		youtube	bbc	linkedin	yahoo	amazon	qq	twitter	taobao	wikipedia	facebook	average
size (KB)		120	152	208	308	320	332	396	476	504	708	352
acc (%)	pipeline	99.9	99.9	99.9	99.9	99.9	99.9	100.0	99.9	99.9	99.9	99.9
	data level	100.0	89.9	66.7	66.7	100.0	33.3	100.0	88.9	85.7	77.8	77.5

Table V. Evaluation Platforms

	MacBook Pro	Nexus 7	Linux Server
CPU	Intel Core i7	Nvidia Tegra 3	Intel Xeon E7
Number of cores	4 cores	4 cores	10 cores
Last-level shared cache	8MB	1MB	20MB
OS	OS X 10.7	Android 4.1	openSUSE 12.1
JVM	HotSpot Server	Dalvik	OpenJDK Sever

become explicit. But our experiments show that in practice, the refinement is quite effective in fixing speculation errors, as shown by the results in the next section.

5. EVALUATION

We implemented both the pipelining and data-level parallelization (HPar) in *jsoup* [jso], an open-source stand-alone HTML5 parser that implements the *HTML5 spec*. This actively maintained open-source parser is written in Java. It is the most popular open-source stand-alone HTML parser in the community.[4]

Benchmarks. We use real HTML pages as our benchmarks. They are downloaded from popular Web sites [top], including *youtube.com*, *bbc.co.uk*, *linkedin.com*, *yahoo.com*, *amazon.com*, *qq.com*, *taobao.com*, *wikipedia.org*, and *facebook.com*. From each of them, we collect one page that contains some substantial content.[5] These pages are mostly the first page showing up after logging into the Web site. Their sizes range from 120KB to 708KB (352KB on average) as shown in Table IV.

Platforms. We use three types of devices with different degrees of portability: a laptop, a tablet, and a Linux server, as listed in Table V.

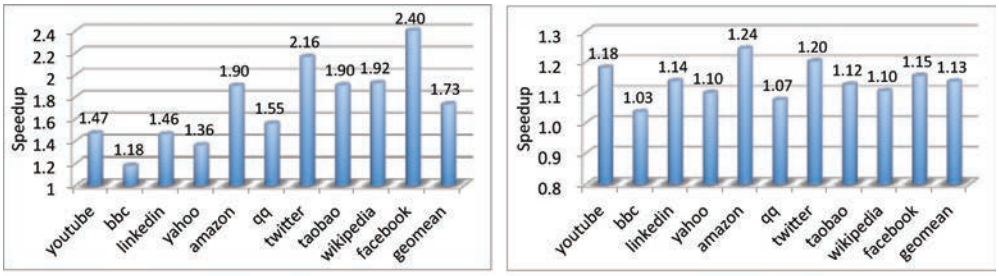
Methodology. To measure the steady-state performance, we repeated each run 10 times after a warm-up run on the Java runtime. We observed some variations among the running times, especially on the portable devices due to the more prominent influence of its many system activities. We used the shortest time for all versions (including the baseline sequential version) in their repetitive runs. The reason for such a policy is that our focus is on the evaluation of the effectiveness of the parallelization techniques. In these runs, the interference from system-level noises is minimal; the time can hence better reflect the actual effects of the parallelization techniques. All speedups are for the parsers' execution time.

5.1. Speculative Data-Level Parallelization

Figure 11(a) reports the speedups of our speculative data-level parallel parser, HPar, on MacBook Pro. The baseline is the default sequential parser from *jsoup*. The rightmost bar reports the geometric mean of the results in all benchmarks.

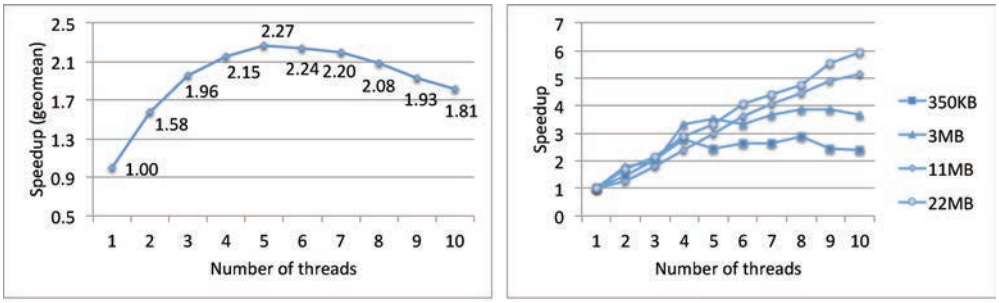
On the MacBook Pro with four cores, HPar achieves speedups as high as 2.40x. Its average speedup is 1.73×. Our analysis shows that the sublinear speedup is due to three main resources. The first is the tree merging step. As a sequential step, it takes about

[4]Its popularity is reflected by the large amount of discussion on it in http://www.stackoverflow.com.
[5]The whole set of web pages is available by emailing requests to the authors.



(a) MacBook Pro (b) Nexus 7

Fig. 11. HPar speedup.



(a) Speedup scalability (b) Impact of input size on scalability

Fig. 12. Scalability of HPar.

5% of the overall parse time in a eight-thread case on MacBook Pro (and 25% on Nexus 7 with four threads). The second is the overhead in the parallelization, including thread creation and communication, and the rollback overhead when the speculation fails. Our speculation success rate is 78% on average, shown in Table IV. The benchmark qq has the largest speculation error because the majority of the cutting points happen to fall into Javascript code embedded in it. Wrong speculations cause some part of input to be reparsed, which generates some, but not much, overhead because the parser realizes and fixes the speculation error quite early such that the reparsed segments are on average only 4.6% of a chunk of input. Yahoo is an exception. Two chunks of it are entirely Javascript code, and the speculation errors were not realized until the end. But Javascript sections do not need to be reparsed by HTML5 parsers. Thus, the errors did not incur much overhead. The third source of overhead is the contention in system resources (e.g., shared cache, memory controllers). Considering that the first two sources typically do not increase much when the input size increases, we expect that when the input becomes larger, the speedup would increase as well. It is confirmed by the scalability study described next.

Scalability Study. Due to the limited number of cores on current portable devices, we conduct the scalability study on a dedicated server equipped with 10 cores as listed in Table V. Figure 12(a) shows the average speedup of the 10 benchmarks as the number of cores grows. The peak speedup appears when five cores are used. An analysis of the individual benchmarks shows that the *facebook* benchmark is the one that has the best scalability, with the speedup up to 2.93×. Considering that *facebook* is the largest benchmark of all, we speculate that the benchmark size could be an important factor for scalability. To confirm it, we create a series of Web pages of a spectrum of sizes by duplicating some Web pages multiple times. Figure 12(b) reports the speedups of

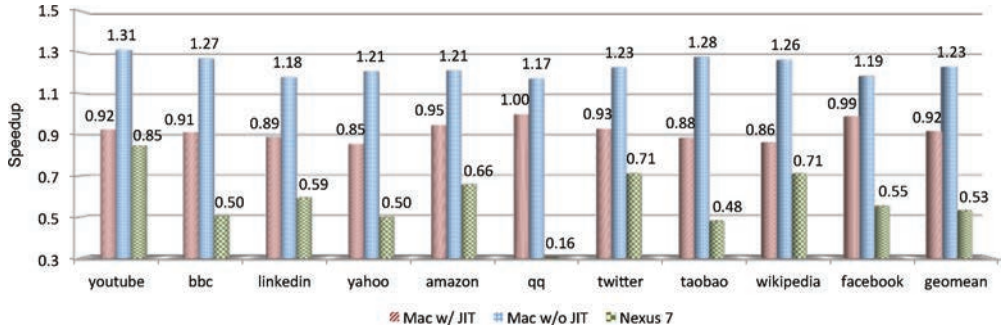


Fig. 13. Speculative pipelining speedup.

HPar on these Web pages when different numbers of cores are used. The scalability increases as the input grows, confirming our speculation. Considering the trend is that Web pages are becoming increasingly complex and large, the results suggest that even larger benefits of HPar may be expected for future Web pages.

The speedup on the Nexus 7 tablet is reported in Figure 11(b). It is relatively modest: up to 1.24× with an average of 1.13×. System resource contention in the parallel runs is the main reason for the speedup to be lower than on the MacBook Pro. To confirm it, we create an artificial scenario that has full parallelism and little parallelization overhead by running four stand-alone copies of the sequential parser on the device. We see 2.3× slowdown compared to the performance of the sequential parser when it runs alone. The larger impact of resource contention is due to the much more limited L1 and L2 cache and memory bandwidth on Nexus 7 than on MacBook Pro (e.g., 1MB vs. 8MB of L2 cache). As handheld devices are evolving rapidly, we expect that the speedup will get closer to that on MacBook Pro as these devices become more powerful in the near future.

5.2. Speculative Pipelining

We have also measured the benefits of the speculative pipelined HTML5 parser. We found that its benefits are much less than HPar. Figure 13 shows the speedup (or slowdown when the value is lower than 1) on the 10 benchmarks. The baseline is also the default sequential parser from *jsoup*. There are three bars for each benchmark. The first one is the speedup on the MacBook Pro. There is no speedup, but an average 8% slowdown. Load balance is not a main issue, as the two stages of the pipeline appear to run for a similar amount of time for most benchmarks. Speculation error is not an issue either, as shown in Table IV. Our analysis show two main reasons for the slowdown. First, the pipeline has only two stages and hence has limited parallelism. Second, there is some overhead for threads creation and synchronization and scheduling, among which the main overhead turns out to be the data transfer from the tokenization thread to the tree builder thread. We tried to change the transfer to a batch fashion rather than in each token but observed no performance benefits. We also tried a spectrum of token buffer sizes to measure its impact on the performance. Figure 14 shows the geometric means of the parsing times of all Web pages when different buffer sizes are used. There is no clear correlation between the parsing time and the buffer size, offering the supporting evidence for using 1-token buffer in our design.

To further analyze the trade-off between time savings and overhead, on the same platform, we conduct another set of experiments. The Java runtime on the laptop allows two modes of code translation, through either an interpreter or the adaptive Just-in-Time (JIT) compiler. By default, the JIT is enabled, as it usually provides

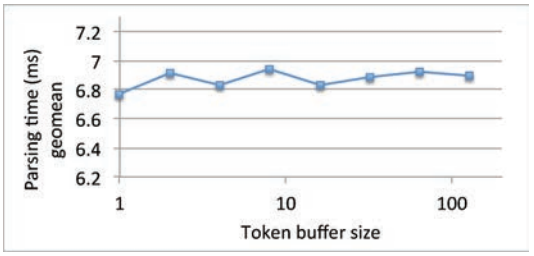


Fig. 14. Performance impact of token buffer size.

better code and hence higher performance (as done in the experiment described in the previous paragraph.) However, in this experiment, we deliberately disable the JIT to see the changes. The second bar in Figure 13 for each benchmark shows the speedups of the pipelined parser compared to the sequential one when JIT is disabled for both. The pipelined parser shows an average 1.29× speedup. Such a dramatic change is because without JIT compilation, the parser runs for a much longer time. Thus, the memory accessing overhead becomes less critical. In other words, the parsing becomes more CPU bound, and hence the savings by the parallelization outweigh the overhead.

The comparison between these two scenarios is mainly to confirm that the pipelined parser works properly and to help understand the trade-offs between its savings and overhead. For practice, the results suggest that the pipelined HTML parser is not beneficial on the MacBook Pro—despite the speedups when JIT is disabled, the fastest version is still the default sequential parser with JIT enabled.

The third bar of each benchmark shows the speedup on the Nexus 7 tablet. The Java runtime is Dalvik with JIT. Considering that the memory bandwidth of the tablet is even smaller than the MacBook Pro, it is expected to see even lower performance of the pipelined parser, given the memory-bound property of the program. The results confirm the expectation: greater slowdowns are shown in the experiments.

Overall, the results indicate that the pipelining parallelization is not a viable way to develop parallel parsers for HTML.

6. POTENTIAL BENEFITS BEYOND PARSING

The benefits of parallel HTML5 parsing go beyond parsing itself. It may open up new optimization opportunities for modern Web browsers. Here we list two examples.

Parallel Object Downloading. HTML documents usually contain various objects, such as images, stylesheets, and Javascript code. These objects are usually not downloaded until the parser reaches the links to those objects in the HTML document. In HPar, the parsing threads start at multiple points in an HTML document; these objects can hence be detected earlier and downloaded speculatively in parallel. A complementary approach to enabling early downloading is prescan [Cascaval et al. 2013], which scans and parses the input at a high level. It has been a sequential process, suffering substantial overhead on large Web pages. The techniques of HPar may be combined with prescan—such as creating a parallel prescan—to minimize the latency.

Parallel Javascript Execution. In our current implementation, the Javascript code in a Web page is executed in the default manner—typically sequentially. But as multiple pieces of Javascript code can now be recognized in parallel and objects embedded in different parts of a Web page could be downloaded concurrently, it is possible for several

pieces of Javascript code in a Web page to be launched in parallel. However, there are some complexities in this optimization. A piece of Javascript code may depend on the parsing threads or on another piece of Javascript code. Hence, to capitalize on these opportunities, one must not ignore these subtle issues.

7. RELATED WORK

7.1. Parallel Browsers

Recent years have seen some increasing interest in parallelizing Web browsers. Web browsers include many kinds of activities. Prior efforts have concentrated on creation of Web page layout [Badea et al. 2010; Meyerovich and Bodik 2010], lexing [Jones et al. 2009], Javascript translation and execution [Intel Corporation n.d.], Web page loading [Wang et al. 2012], XML parsing [Lu et al. 2006; Wu et al. 2008], and Web page decomposing [Mai et al. 2012]. Industry (e.g., Qualcomm and Mozilla) has begun to design new browser architectures [Mozilla Corporation n.d.; Cascaval et al. 2013] to leverage these results or better exploit task-level parallelism. These studies have not focused on parallelizing HTML parsing, an important component for the response time of a browser. Integration of HPar into a browser is ongoing work.

7.2. Parallel Parsing

Many studies have been devoted to parallel parsing between the 1970s and 1990s. They fall into two classes: parallel parse of programming languages or natural languages. For programming languages, arithmetic expressions and bracket matching are mostly studied and evaluated. A pipeline is used between lexing and parsing [Pronina and Chudin 1975], or even among lexing, parsing, and semantic processing [Baer and Ellis 1977]. For parsing itself, the parallelism can come from either grammar decomposition or input decomposition [Alblas et al. 1994]. Fischer's [1975] seminal work proposes a parallel parsing algorithm based on LR parsing. Given a starting point of the string, the algorithm uses a number of sequential parsers for every possible parsing state and maintains a parsing stack for each parser. Prefix-sum algorithm is also applied to parallel parsing problems to compute the effects of a series of stack transitions [Skillicorn and Barnard 1989]. These approaches suffer from scalability issues for the rapid growth of the possible states and transitions of the stack as the vocabulary and inputs increase. In grammar decomposition, the grammar is decomposed such that each processor is in charge of a part of the grammar [Baccelli and Fleury 1982; Baccelli and Mussi 1986]. The decomposition exploits limited parallelism compared to input data decomposition. Luttighuis [1989] leveraged parallel bracket matching algorithms for parallel parse of some special subclasses of context free language. In natural language processing, many efforts have been put to CYK [Kasami 1965; Younger 1966; Yi et al. 2011] and Earley [1970] parsing algorithms. A recent work has examined how to synthesize a parallel schedule of structured traversals over trees, offering some help for parallel CSS processing [Meyerovich et al. 2013]. These previous studies have laid the foundation for this work. However, most of them have focused on languages defined with clean formal grammars. They have not been able to be applied to HTML successfully, as Section 2 has discussed. Natural languages are also defined in an ad hoc manner. However, they have a whole set of different complexities from HTML. The parallel parsing techniques developed for natural languages have a high time complexity, unsuitable for programming languages.

Parallel XML Parsing. The two basic ways of data-level parallel parsing—the partition oriented and the merging oriented—have been applied for XML parallel parsing [Shah et al. 2009; Pan et al. 2007; Lu et al. 2006; Wu et al. 2008]. The work by Lu et al. [2006], for example, first extracts the high-level structure of XML documents through a quick prescan and parses each part of the document in parallel. The work by

Wu et al. [2008] cuts XML documents into chunks, parses them, and merges the result together. HTML5 parsing has many special complexities compared to XML parsing. In fact, all of the major complexities listed in Section 2.2 are not presented in typical XML documents and hence have been ignored by previous studies in XML parallel parsing. For example, none of them handles error correcting, embedding of other languages, and executable scripts that modify the document in which it resides. As we discussed in Section 4.2, these complexities cause complicated effects on the data dependencies in HTML5 parsing and hence the many difficulties for parallelization. They prompt our development of the speculative techniques described in this work. In contrast, simple methods have been used in prior XML parallel parsing studies. The work by Wu et al. [2008] simply cuts input right after a right angle bracket and lets every thread parse their chunk from the initial state. They do not consider the failing cases, in which the cutting point is actually inside a string of comment, in the middle of a piece of Javascript code, or after an erroneous ending tag. They do not have the sophisticated speculation designs or error-handling schemes as we present. Nor do they handle modifications of the document by the scripts it contains. It is exactly these special complexities that have been the key barriers for HTML parallel parsing. Effectively handling them to enable HTML parallel parsing is the distinctive contribution of this current work.

There are some prior studies on using context to help parse languages that allows embedding of domain-specific languages [Wyk and Schwerdfeger 2007]. They are for enhancing sequential parsers rather than parallelization.

To the best of our knowledge, this work is the first study that systematically investigates the special complexities in parallel HTML parsing; by developing a set of speculation-centered solutions customized to HTML parsing, it creates HPar, the first practical parallel parser for HTML.

7.3. Speculative Execution

A number of studies have explored speculative execution techniques for solving challenging parallelization problems at both the levels of programming languages and compilers [Prabhu et al. 2010; Ding et al. 2007; Feng et al. 2011; Raman et al. 2010; Quiones et al. 2005] and architecture designs [Marcuello and González 2002; Colohan et al. 2002]. To simplify programmers' task, Prabhu et al. [2010] proposed two new language constructs to parallelize applications with speculation. Some work uses software speculation to parallelize certain regions of the applications. This can be implemented either at process level [Ding et al. 2007] or thread level [Feng et al. 2011; Raman et al. 2010]. These automatic or semiautomatic approaches for general application parallelization have their limitations. They usually focus on loops, and the dependencies with which they can deal are relatively simple. A recent study introduces rigorous analysis to DFA speculation [Zhao et al. 2012].

8. CONCLUSION

This article presents the first systematic study in taming the complexities of HTML5 and developing speculation-centered techniques to create parallel HTML parsers. The outcome includes a set of insights on effectively parallelizing HTML parsers, as well as HPar, the first practical HTML parallel parser that yields up to 2.4× speedup (1.73× on average) on quad-core devices. This study breaks a challenging barrier for fully parallelizing Web browsers, improves the efficiency of a critical component in modern Web browsers, and opens up new opportunities for browser optimizations.

ACKNOWLEDGMENTS

We owe the anonymous reviewers of this work and its earlier versions our gratitude for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant Nos.

0811791 and 1320796, the CAREER Award, and the DOE Early Career Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or DOE.

REFERENCES

HTML Living Standard: Section 12.2 Parsing HTML Documents. http://www.whatwg.org/specs/web-apps/current-work/multipage/parsing.html.

HTML reference. http://www.w3schools.com/tags.

Jsoup: Java HTML Parser. http://jsoup.org.

The HTML5 test. http://html5test.com/.

Top 1,000 Web sites. http://www.alexa.com/topsites.

Validator.nu. http://about.validator.nu/.

Web pages getting bloated here is why. Retrieved from http://royal.pingdom.com/2011/11/21/web-pages-getting-bloated-here-is-why/.

Alblas, H., op den Akker, R., Luttighuis, P. O., and Sikkel, K. 1994. A bibliography on parallel parsing. *SIGPLAN Not. 29,* 1, 54–65.

Baccelli, F. and Fleury, T. 1982. On parsing arithmetic expressions in a multiprocessing environment. *Acta Inf. 17*, 287–310.

Baccelli, F. and Mussi, P. 1986. An asynchronous parallel interpreter for arithmetic expressions and its evaluation. *IEEE Trans. Computers 35,* 3, 245–256.

Badea, C., Haghighat, M. R., Nicolau, A., and Veidenbaum, A. V. 2010. Towards parallelizing the layout engine of Firefox. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar'10)*.

Baer, J. L. and Ellis, C. S. 1977. Model, design, and evaluation of a compiler for a parallel processing environment. *IEEE Trans. Softw. Eng. 3,* 6, 394–405.

Cascaval, C., Fowler, S., Montesinos-Ortego, P., Piekarski, W., Reshadi, M., Robatmili, B., Weber, M., and Bhavsar, V. 2013. Zoomm: a parallel web browser engine for multicore mobile devices. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY, 271–280.

Colohan, C. B., Zhai, A., and Mowry, T. C. 2002. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High Performance Computer Architectures*.

Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., and Zhang, C. 2007. Software behavior-oriented parallelization. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*.

Earley, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM 13,* 2, 94–102.

Schurman, E. and Brutlag, J. 2009. Performance related changes and their user impact. Retrieved from http://velocityconf.com/velocity2009.

Feng, M., Gupta, R., and Hu, Y. 2011. Spicec: Scalable parallelism via implicit copying and explicit commit. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*.

Fischer, C. N. 1975. *On Parsing Context Free Languages in Parallel Environments*. Ph.D. thesis, Cornell University.

Hoxmeier, J. and Dicesare, C. 2000. System response time and user satisfaction: An experimental study of browser-based applications. In *Proceedings of the Association of Information Systems Americas Conference*.

Intel Corporation. n.d. River Trail. Retrieved from https://github.com/RiverTrail/RiverTrail.

Jones, C. G., Liu, R., Meyerovich, L., Asanović, K., and Bodík, R. 2009. Parallelizing the web browser. In *Proceedings of the 1st USENIX Conference on Hot topics in Parallelism (HotPar'09)*.

Kasami, T. 1965. *An Efficient Recognition and Syntax Analysis Algorithm for Context-free Languages.* Technical Report AFCRL-65-758. Air Force Cambridge Research Laboratory.

Kohavi, R. and Longbotham, R. 2007. Online experiments: Lessons learned. *IEEE Computer 40,* 9, 103–105.

Lu, W., Chiu, K., and Pan, Y. 2006. A parallel approach to XML parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID'06)*. 223–230.

Luttighuis, P. 1989. *Parallel Parsing of Regular Right-part Grammars*. Memoranda Informatica.

Mai, H., Tang, S., King, S. T., Cascaval, C., and Montesinos, P. 2012. A case for parallelizing web pages. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*. 2–2.

Marcuello, P. and González, A. 2002. Thread-spawning schemes for speculative multithreaded architectures. In *Proceedings of the 8th International Symposium on High Performance Computer Architectures*.

MEYEROVICH, L. A. AND BODIK, R. 2010. Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*.

MEYEROVICH, L. A., TOROK, M. E., ATKINSON, E., AND BODIK, R. 2013. Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. 187–196.

MICHAEL, M. M. AND SCOTT, M. L. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Ssymposium on Principles of Distributed Computing (PODC'96)*. 267–275.

MOZILLA CORPORATION. n.d. Servo. https://github.com/mozilla/servo.

NAH, F. 2004. Study on tolerable waiting time: How long are web users willing to wait? *Behavior and Information Technology 23*, 3, 153–163.

PAN, Y., LU, W., ZHANG, Y., AND CHIU, K. 2007. A static load-balancing scheme for parallel XML parsing on multicore CPUs. In *Proceedings of the 7th International Symposium on Cluster Computing and the Grid (CCGRID'07)*.

PRABHU, P., RAMALINGAM, G., AND VASWANI, K. 2010. Safe programmable speculative parallelism. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*.

PRONINA, V. AND CHUDIN, A. 1975. Syntax analysis implementation in an associative parallel processor. *Automation and Remote Control 36,* 8, 1303–308.

QUIONES, C., MADRILES, C., SNCHEZ, F. J., MARCUELLO, P., GONZÁLEZ, A., AND TULLSEN, D. M. 2005. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation.*

RAMAN, A., KIM, H., MASON, T. R., JABLIN, T. B., AND AUGUST, D. I. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

SHAH, B., RAO, P., MOON, B., AND RAJAGOPALAN, M. 2009. A data parallel algorithm for XML DOM parsing. *Database and XML Technologies, Lecture Notes in Computer Science 5679*.

SKILLICORN, D. B. AND BARNARD, D. T. 1989. Parallel parsing on the connection machine. *Inf. Process. Lett. 31,* 3, 111–117.

VACHHARAJANI, N., RANGAN, R., RAMAN, E., BRIDGES, M. J., OTTONI, G., AND AUGUST, D. I. 2007. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. 49–59.

WANG, Z., LIN, F. X., ZHONG, L., AND CHISHTIE, M. 2012. How far can client-only solutions go for mobile browser speed? In *Proceedings of the 21st International Conference on World Wide Web (WWW'12)*. 31–40.

WU, Y., ZHANG, Q., YU, Z., AND LI, J. 2008. A hybrid parallel processing for XML parsing and schema validation. In *Proceedings of Balisage: The Markup Conference 2008*.

WYK, E. AND SCHWERDFEGER, A. 2007. Context-aware scanning for parsing extensible languages. In *Proceedings of the International Conference on Generative Programming and Component Engineering*.

YI, Y., LAI, C.-Y., PETROV, S., AND KEUTZER, K. 2011. Efficient parallel CKY parsing on GPUs. In *Proceedings of the 12th International Conference on Parsing Technologies (IWPT'11)*. 175–185.

YOUNGER, D. H. 1966. Context-free language processing in time n3. In *Proceedings of the 7th Annual Symposium on Switching and Automata Theory*. 7–20.

ZHAO, Z., WU, B., AND SHEN, X. 2012. Speculative parallelization needs rigor: Probabilistic analysis for optimal speculation of finite state machine applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

Received June 2013; revised September 2013; accepted November 2013