Divide-and-Conquer Parsing for Parallelism and Laziness

by Seth Fowler

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor R. Bodík
Research Advisor

(Date)

* * * * * * *

Professor E. Brewer
Second Reader

(Date)

1. Introduction

Today is the era of web applications. Google alone has brought us Gmail, Google Maps, Google Docs, and Google Calendar, all wildly popular. Similar offerings exist from Yahoo, Microsoft, Apple, Zoho, and a host of other software vendors. And of course we cannot forget Facebook, YouTube, or Flickr.

The convenience the browser has brought to computing is hard to overstate. Applications are whisked from faraway datacenters over the Internet and onto our machines in the time it takes to click a link. Data is effortlessly in sync and available from anywhere. From the user's perspective, software versions and upgrades have simply ceased to exist. Web apps just work, regardless of your operating system or processor architecture. The browser has delivered on the thin client promise, and computing is more flexible, portable, and low-maintenance than ever before. Soon, we are told, we'll do everything in the browser.

But if web applications offer all of these benefits, then why do native applications exist at all anymore? There are a number of factors: the work involved in porting large codebases to a new platform and language can be tremendous, and the web's reduced potential for vendor lockin encourages a certain degree of inertia on the part of incumbent players. Browser incompatibilities, user reluctance to upgrade, and slow adoption of new standards make things difficult even for new applications. If all these problems were overcome, though, it still wouldn't be feasible to write a credible competitor to Adobe Photoshop, Autodesk Maya, or Ableton Live on the web application stack available today, because the performance required is just not available.

The browser doesn't have just one performance problem, it has many. Runtime performance of web applications is restricted by the difficulty of optimizing JavaScript, by limitations on parallelism imposed by the shared nature of the DOM, and by lack of access to hardware acceleration features enjoyed by native applications. Startup performance is impaired by the realities of application delivery over a shared network and JavaScript's delivery in source, rather than compiled, form. For web applications to replace native applications, we'll need to solve or mitigate all of these problems.

In this report, we propose divide-and-conquer parsing as a technique to mitigate the impact on web application startup time caused by the need to lex, parse, analyze, and compile JavaScript at page load time. Web standards require that page rendering pause when JavaScript code is encountered because its execution can affect the context that the rest of the page is rendered in. JavaScript can thus have a significant negative effect on page load time. Divide-and-conquer parsing can greatly reduce this effect by partitioning JavaScript code into small, independent chunks which can be parsed (and potentially analyzed and compiled) in parallel or on-demand. The result is that JavaScript code can begin executing sooner, the browser is able to resume rendering the page more quickly, and the web application becomes responsive to the user faster.

This report makes the following main contributions:

- Divide-and-conquer parsing, a generalization of existing techniques for parsing text out-of-order
- A parallel divide-and-conquer parser capable of delivering nearly 3x speedup on a machine with four hardware threads in our tests

We also describe some other applications of divide-and-conquer parsing:

- Lazy parsing, a technique for improving parsing performance by avoiding work that isn't immediately useful
- Speculative parsing, which extends lazy parsing by predicting which work may be useful in the near future, allow laziness to be combined with parallelism

The remainder of the report is organized as follows. We review background information and discuss the limitations of traditional parsers that we hope to address in section 2. Divide-and-conquer parsing is introduced in section 3, and it is used to implement a parallel parser in section 4. We evaluate an implementation of this design in section 5. After discussing lazy parsing and speculative parsing, other techniques which leverage divide-and-conquer parsing, in section 6, we review related work in section 7. We summarize the contributions of this report and conclude in section 8.

2. Background

To provide context for the following discussion, in this section we review the parsing problem, the limitations of conventional parsers, and the design constraints that led to our proposed solution. Section 2.1 summarizes the basics of parsing. We then describe the in-order nature of efficient parsing algorithms and show the difficulties involved in out-of-order parsing in section 2.2. We conclude this section by exploring ways these difficulties can be overcome and the tradeoffs of each approach in section 2.3.

2.1 Parsing

Parsers are programs that read a sequence of implicitly structured symbols and produce a data structure that contains the same information in an explicitly structured form that is easier for a computer to manipulate. Conventionally, the symbols represent text — they are either characters or words. They are imparted an implicit structure because the parser interprets them according to the syntax of some language, such as English, first-order logic, or JavaScript.

A language's syntax is described using a recursive system of rules called a grammar. If we identify a language with the set of strings that belong to that language — for example, if we identify English with the set of all valid English documents — then we can think of a grammar as a concise description of that set which gives rules for generating its elements instead of listing them explicitly. To generate an element, one begins with a string containing only an initial symbol (termed the grammar's start symbol) and repeatedly applies the rules of the grammar. Each rule is of the form α `::=` β, where α and β are sequences of symbols; such a rule means that when the string contains α, α can be replaced by β. There can be more than one rule with α on the left-hand side, which can be denoted using the short-hand notation α `::=` β `|` γ; in this case, the rule to apply can be chosen arbitrarily, allowing a single grammar to generate many strings. Recursion occurs when β contains α or can generate α through repeated rule application; a recursive grammar can describe an infinitely large language. The process of generating a string is complete when no more rules can be applied, which occurs when the string contains no substring that appears on the left-hand side of any rule.

A parser may use the grammar of a language to impose structure on a string of symbols using two main strategies. In top-down parsing, the parser simulates the process of generating a string in the language. When there are multiple ways to proceed, the parser makes a choice that is consistent with the sequence of symbols. The parser's choices can be summarized in a tree where an edge exists from an α in one version of the generated string to the each symbol in the β that replaces it in a later version. This tree is called a parse tree; if the αs in the grammar are chosen to describe meaningful syntactic units of the language (for example, the subject and direct object of a sentence in English) then the parse tree conveniently describes the structure of the string it was created from. Bottom-up parsing also generates such a parse tree, but it works by looking for βs and replacing them with the corresponding αs until the grammar's start symbol is reached. The tree is thus constructed from the leaves rather than from the root — hence the term "bottom-up".

Though both parsing strategies naturally produce parse trees, the larger program making use of the parser may prefer a somewhat different representation — for example, when parsing a mathematical expression it is useful to remove parentheses from the parse tree, since the nesting they serve to express is already encoded in the form of the parse tree itself. Parsers thus make use of "semantic actions", functions associated with each α in the grammar that take an α vertex in the parse tree and return a domain-specific representation of that vertex.

The parsing concepts we have describe so far are very generic and much more powerful than we need in practice for parsing computer languages like JavaScript. We will restrict ourselves for the remainder of this report to grammars with two special characteristics. First, each α must consist of only one symbol, termed a nonterminal, which is not among the symbols found in the language itself, termed terminals; if a grammar has this property, it is described as "context-free". Second, each string in the language must be generated by the grammar in exactly one way — that is, there must be a one-to-one mapping between strings in the language and parse trees produced by following the rules of the grammar — and the parser must be able to determine the appropriate choices to generate the string while only considering a fixed number of symbols per choice; a grammar with this property is described as "deterministic". Deterministic context-free grammars can be parsed efficiently,[1] and in practice most computer languages of interest can be described by such grammars, including the languages used on the web like HTML and JavaScript.

2.2 In-Order Parsing

Efficient parsing algorithms designed for use with deterministic context-free grammars are traditionally in-order: they read symbols from an input stream, interpret them in light of their internal state, update that state, and invoke an appropriate semantic action to produce whatever sort of output is required by the larger application. The in-order approach is simple and quite efficient, and deviating from it can be costly in terms of performance. Consider the grammar in Listing 1:[2]

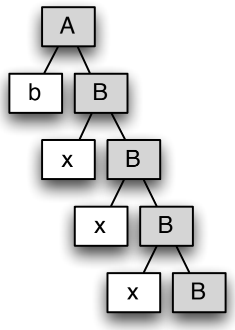
```
A ::= b B | c C
B ::= x B | x
C ::= x C | x
```

Listing 1: A simple grammar.

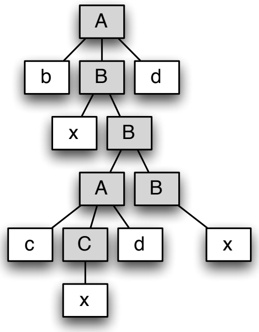
This grammar generates the language $\{bx^+ \cup cx^+\}$ — that is, all strings starting with a b or a c followed by one or more xs. An application may intend different semantics for the xs generated by B than for those generated by C, so it is important that the parser recognize which kind of x it is faced with and execute the appropriate semantic action. This will happen naturally if the parser moves through the input in order. Consider the input `bxxx`, with the parse tree shown in Figure 1a. The parser can see that the initial b symbol must have been generated by the rule `A ::= b B` and will parse the succeeding xs using the rule for B.

[1] LR parsers can be used with any such grammar and run in time quadratic in the length of the input.

[2] We use the convention that the start symbol is the left-hand side of the first rule.



(a) `bxxx`



(b) `bxcxdxd`

Figure 1: Parse trees.

Imagine, however, a parser that does not move through the input in order; it might instead split the input into two fragments, `bx` and `xx`, and attempt to parse `xx` first. It will immediately run into a problem: how should the x symbol at the beginning of `xx` be interpreted? It could have been generated by the rule for B or the rule for C, and there is no information in this fragment that can resolve the question. If the parser is allowed to read two characters backwards in the input, into the previous fragment, it can find the b symbol and understand how to proceed, but in the general case it might have to read arbitrarily far, since there is no limit to the size of the strings in this language. In this particular case the decision can be made by examining the first symbol in the input, but things can get worse:

```
A ::= b B d | c C d
B ::= x B | A B | x
C ::= x C | A C | x
```

Listing 2: A grammar with nesting.

The grammar in Listing 2 allows nesting. Consider the input `bxcxdxd`; the corresponding parse tree is shown in Figure 1b. In-order parsing works in just the same way as before, but parsing the input in arbitrary order is even more difficult here. Imagine that we split the input into two fragments again: `bxcxd` and `xd`. How does the parser know how to interpret the x symbol at the beginning of `xd`? It can no longer simply check a symbol at a certain offset, like it could with the previous grammar. It can't even read backwards in the input until it finds a b or a c; here it would first encounter the c in `bxcxd`, suggesting that the x symbol was generated by the C rule, but we know this is wrong because the d at the end of the first fragment is generated after a C rule has been completely expanded.

To interpret the x symbol correctly, the parser must maintain a stack as it reads backwards, pushing each d symbol onto the stack and popping the stack when it encounters a b or a c, until it encounters a b or a c while the stack is empty. Following this approach, the parser pushes a d, pops it when it sees the c, and arrives at the b at the beginning of the first fragment with an empty stack. This means that the x symbol at the beginning of `xd` was generated by the B rule.

In the general case reading backwards and manipulating a stack like this is a lot of work, just to determine information that the in-order parser gets "for free". We may end up essentially parsing the first fragment backwards before we can parse the second one forwards, in which case it would clearly have been better just to parse the fragments in order in the first place. Even worse, imagine that the input had contained an additional level of nesting, so that the first fragment became `bxcxbxd` and the second became `xdxd`.

Now there are two ambiguous xs in the second fragment, and we have to work backwards *twice* to interpret them both correctly.

To parse fragments of the input in a different order than the usual one, then, can be an expensive proposition if done naïvely. However, solving this problem will allow us to parse each fragment independently, in parallel or on-demand. We explore improvements over the naïve approach in the next section.

2.3 Out-of-Order Parsing

Parsing fragments of an input in arbitrary order requires that we be able to reconstruct each fragment's "initial state" — that is, the state that an in-order parser would have had at the beginning of the fragment. This state takes the form of knowledge about which rules generated some or all of the current fragment, beginning in an earlier fragment. Since there is no limit to the nesting depth recursive rules may generate, the size of an initial state is unbounded in real-world languages. We therefore refine the concept by noting that we need only include information that is actually necessary to parse the current fragment.

Since our focus is on deterministic context-free grammars, it's worth making this concrete in terms of the bottom-up *LR*[22] parsers which are most often used to parse such grammars. In LR parsers, state information is encoded in a stack; the top entry in the stack represents the set of rules that could have most directly generated the symbols at the current position in the input, the next entry represents the set of rules that could have generated those rules, and so on. Thus, a fragment's initial state in an LR parser consists of those n top-most elements on the stack which must be consulted to successfully parse the fragment.

2.3.1 Naïve Out-of-Order Parsing

In the previous section, we discussed a naïve approach to parsing out-of-order: work backwards from the start of the fragment until the parser can uniquely determine the initial state. This method can require, in the worst case, reading all the way back to the beginning of the input, unneccessarily duplicating much of the work that will be done to parse other fragments along the way.

2.3.2 Speculative Out-of-Order Parsing

Another alternative is to simply try every feasible initial state, producing multiple parses for the same fragment. Once the final state of the preceding fragment (and, transitively, every earlier fragment) is known, the correct alternative can be selected, and its parse tree can be joined to that produced by the earlier fragments. This approach allows each fragment to be parsed independently as long as a cleanup step is performed at the end, but like the naïve approach, it can be expensive.

Consider an LR parser. Every time the parser empties its stack while parsing a fragment, it suddenly has no knowledge about which rules could have generated the input it's seeing, and must guess what would have been on top of the stack if it had access to the fragment's initial state. It must always guess among at least two possibilities — if there is only one possibility, after all, it's not a guess. Since the parser's guess can affect the interpretation of the rest of the fragment, each sequence of guessed choices forms a new parse tree, and the number of such parse trees is exponential in the number of guesses the parser must make.

Also problematic for this scheme are the limitations it places on semantic actions; if the parser executes semantic actions while constructing speculative parse trees, then the semantic actions must not have any global effects (which are common for programming languages which often track things like variables and types in tables outside the parse tree) since there is no way to know at this point which semantic actions are correct with respect to the true initial state of the fragment.

There are some ways to mitigate these problems. The space requirements of storing an exponential number of parse trees can be made linear in the length of the fragment if the parser records only information about which rules could have generated which portions of the input.[3] Unfortunately, the extra work is still exponential. We can reduce its impact by statistical means — for example, we might try the top n most likely alternatives for each guess, based on statistics from real-world documents — or we can simply bet that most guessed choices will fail quickly, but for our goal of improving performance, exponential additional work seems less than ideal.

2.3.3 Conservative Out-of-Order Parsing

To avoid the costs of this guesswork, we may try a third approach: the parser may scan forward through the fragment, looking for subfragments that can be parsed without knowing the initial state. The rest of the fragment can then be parsed when the final state of the previous fragment is available, with the parse trees of the subfragments merged in at appropriate places. Thus, this approach uses out-of-order parsing for as much of the fragment as possible, and in-order parsing where out-of-order parsing won't work or is too expensive.

2.3.4 Towards Divide-and-Conquer Parsing

We can eliminate in-order parsing totally if we run an initial pass over the input to rewrite it in a form that is more amenable to out-of-order parsing. It happens that LR parsers typically do not operate on symbols that represent characters, but rather use symbols representing larger units of meaning, like keywords in a programming language. Thus, before an input consisting of characters can be parsed by an LR parser, it must usually be translated into the symbols that parser understands. This process is performed by a program called a "lexer". Since the lexer must rewrite the input anyway, we can make it suitable for out-of-order processing without needing to perform any extra passes. To do this, we modify the lexer so that it identifies fragments of the input that can be parsed independently and separates them out. This is performed hierarchically, so that the lexer identifiers fragments not just within the input but also within other fragments. When the lexer finds a subfragment that can be parsed independently within a larger fragment, it stores that subfragment separately and replaces it in the larger fragment with a special "reference symbol".

The reference symbol serves two roles: it identifies the rule that generated the subfragment it replaces, so that the larger fragment can be parsed without inspecting the subfragment, and it contains a reference to the memory location that will store the subfragment's parse tree once it is constructed. With this arrangement, the input is decomposed completely into fragments that can all be parsed out-of-order, and no reassembly step is necessary since the reference symbols supply the edges that would otherwise be missing from the parse tree after each fragment is parsed separately.[4] The fragment generated directly by the start symbol becomes the "root fragment", and the reference symbols ensure that its parse tree becomes the complete parse tree of the input once every fragment has been parsed.

[3] This is the encoding used by chart parsers, such as the Earley and packrat parsers.

[4] The tree formed by the reference symbols is therefore a minor of the original parse tree.

The problem with this final method is implicit in the nature of the reference symbols: since they encode a subset of the edges in the parse tree, we can see that dividing the input into fragments and linking them with reference symbols in this manner must essentially require parsing the input! If we need to parse the input in-order to parse it out-of-order, we haven't accomplished much. However, there is a way out of this predicament: we can divide the input into independent fragments using simple criteria that the lexer can check without performing any parsing. In section 3, we discuss how this can be achieved in general and give a concrete example for JavaScript.

3. Divide-and-Conquer Parsing

Divide-and-conquer parsing ("DAC parsing") is an out-of-order parsing technique appropriate for use with deterministic context-free grammars. A DAC parser divides its input into fragments such that each fragment can be parsed independently using standard, efficient parsing algorithms. The fragments are augmented with metadata in such a way that the same parse tree (or whatever data structure is constructed by semantic actions) can be automatically constructed when parsing the individual fragments, with no separate pass required to join the fragments' subtrees together.

In this section, we'll discuss the details of DAC parsing and how a DAC lexer and parser differ from conventional designs. To make our discussion concrete, we will focus our examples on the JavaScript programming language. Consider the JavaScript program in Listing 3:

```
var a = 1;
var f = function(x)
        {
          function dbl(n) { return n * 2; }
          return x + dbl(x);
        }
var g = function(y) { return y - 2; }
var b = g(f(a));
```

Listing 3: A simple JavaScript program.

This program is so small that almost any parsing technique can parse it efficiently, but it has enough complexity to serve as a useful example input for a DAC parser. As we discuss the transformations the parser applies, we'll illustrate each one with successive transformed versions of this program.

The remainder of this section is organized as follows. Section 3.1 presents the design of a DAC lexer. The details of implementing such a lexer depend on the evaluation strategy chosen for the parser; we explain this choice and its effects in section 3.2. DAC parsers use reference symbols to avoid the need for explicitly combined the results of parsing the individual fragments; details are provided in section 3.3. Section 3.4 concludes with a discussion of the DAC parsing component itself.

3.1 Divide-and-Conquer Lexing

Parsing for deterministic context-free grammars conventionally begins by running the input, in the form of textual characters, through a lexer, which produces a transformed version of the input represented in terms of symbols. These symbols are selected from the terminals of the grammar, and are often augmented with metadata. For example, a grammar for the JavaScript language might use one symbol for all numeric literals, with the metadata indicating the particular number — 2, for example. The actual parsing work is then performed on these symbols, rather than on the original text.

This arrangement is fortuitous for a DAC parser, because it needs to identify independent fragments of the input — that is, fragments that do not require any initial state to parse correctly — before it can parse them. We achieve this without requiring an additional pass over the input by performing this work in the lexer. The lexer is modified to recognize sequences of characters that indicate the beginning and end of independent fragments, and it extracts those fragments to be parsed independently.

But how does the lexer find the fragment boundaries? Lexers are conventionally regular automatons, with the recognition power of a regular expression.[5] This ensures that they are fast, taking time linear in the input, but it also means that they have no notion of recursion and so cannot recognize nested structures. If the fragments we are interested in are not nested inside one another, and the symbols that mark their boundaries cannot also occur inside them, then we don't have a problem. Unfortunately, few languages of real-world interest make things so easy on us. Nesting is omnipresent in programming languages, JavaScript included.

If we were able to decompose JavaScript programs into fragments with no restrictions, we'd like our fragments to be large enough that they make as much work as possible available for semantic actions to do without requiring the inspection of subfragments which may not have been parsed yet. We'd also like our fragments to provide a fine-enough decomposition on real-world inputs that they are useful for applications like parallelism or laziness. Finally, it would be ideal if fragments are generated entirely by the rules for a single nonterminal, so that we can parse them independently with the same grammar by using that nonterminal as a start symbol. For JavaScript, and many other programming languages, functions are a natural fragment granularity given these requirements.

JavaScript functions begin with the `function` keyword followed by a sequence of other symbols which vary depending on the details of the function definition. The sequence always ends in a left brace ({). A right brace (}) at the same nesting level as the left brace terminates the function, but nested left and right braces may occur within the function. For the lexer to recognize functions, then, it must be able to recognize nested constructions. Regular lexers cannot do this, but DAC lexers can, because they make use of a counter.[6] The counter is incremented when a left brace is encountered, and decremented for a right brace. When the lexer encounters the sequence indicating the beginning of a function, it records the current nesting level. Every time it updates the counter, it checks if the nesting level has returned to its value at the start of the function. If so, then the lexer has arrived at the end of the function, and has identified an independent fragment.

In JavaScript, functions can be nested, which means that this simplistic approach isn't quite enough. Indeed, because functions are often used for namespacing in JavaScript, this mechanism will often reduce a JavaScript library to only a couple of fragments. To support hierarchical functions, the DAC lexer uses a "fragment stack". Every time it locates the beginning of a function, corresponding to the beginning of an independent fragment, it pushes the current nesting level onto the stack. On counter updates it checks if the current nesting level is the same as the value recorded on top of the stack; if so, the lexer has arrived at the end of a function, and it pops the stack.

[5] This is not the same as a Perl-style regex, which are considerably more powerful and correspondingly asymptotically slower.

[6] This means that they are no longer regular automatons; they can instead be regarded as nested word automatons.[18]

3.2 Symbol Buffers and Evaluation Strategies

A normal lexer uses one of two approaches for storing the symbols it creates, determined by the evaluation strategy of the parser it is paired with. If it is used with a deferred parser, which won't begin parsing until the entire input is lexed, it stores all of the symbols in the same large buffer in the order they are generated. For an eager parser, which consumes each symbol as soon as it becomes available, it uses no symbol buffer at all.

If a DAC lexer is paired with an eager parser, it too requires no symbol buffer. A deferred parser, however, requires a different approach; the operation of the parser will be much simpler (and much more cache-friendly) if the symbols for each fragment are arranged contiguously in memory. Though a normal lexer cannot support it, a DAC lexer can also be used with a lazy parser, which parses each fragment only when it must do so because the fragment's parse tree is needed. Lazy parsers also benefit if fragments are contiguous.

Unfortunately, hierarchically nested fragments do not naturally arrive arranged contiguously in memory. If the symbols for every fragment are written to memory in the same order they appear in the input text, many fragments will be separated into several pieces scattered through memory. Consider the input-order layout of the JavaScript program in Listing 3, shown in Figure 2. The fragment representing the function `f` is separated into two pieces, and the root fragment is separated into three!

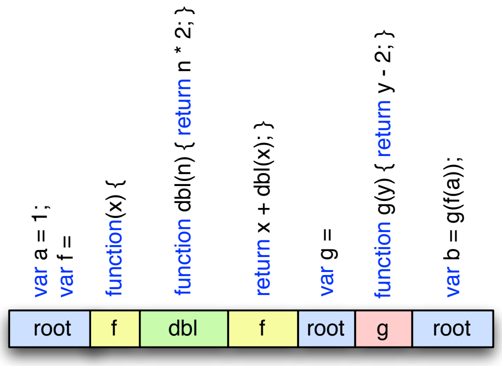


Figure 2: Hierarchically nested fragments are noncontiguous if stored in input order.

To solve this problem, we could dynamically allocate a new buffer to store each fragment into. However, dynamically allocating memory is expensive, and since we cannot know the final size of a fragment until we are done processing it, we have to either perform expensive resizing operations or allocate much more memory than we really need, both of which increase memory fragmentation and harm performance.

We can avoid allocating a buffer for each fragment by noting that because our fragments are properly nested, a new fragment at a given nesting depth will never begin until the previous fragment at that nesting depth has ended. If we allocate one buffer per fragment stack entry, and store the symbols for a fragment into the buffer on top of the stack, every fragment can be contiguous. If the fragment stack entries are retained when the stack is popped and reused the stack is pushed, the number of buffers that must be allocated is equal to the maximum number of nested fragments encountered in the input. Although these buffers will have to be resized occasionally, the usual approach of exponentially growing the buffer on every resize will keep the total number of resizes to a minimum, and copying can be avoided with an appropriate choice of data structure;[7] the same issue and the same solutions arise in the conventional deferred parsing case. With this approach, the code in Listing 3 will have the memory layout shown in Figure 3.

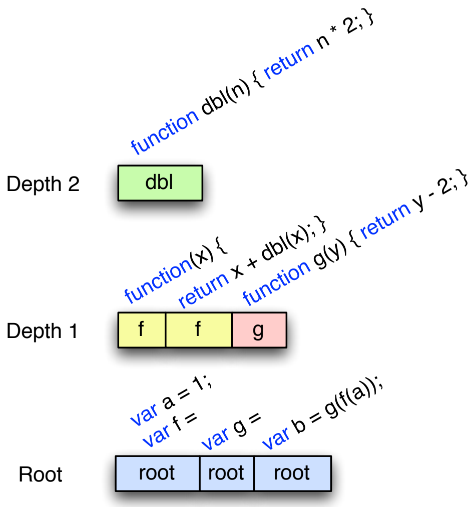


Figure 3: Hierarchically nested fragments are contiguous if fragment stack buffers are used.

3.3 Reference Symbols

When a nested fragment is separated out from its parent, the hole left in the parent fragment cannot simply remain empty. The parser may perceived these holes as syntax errors; even if it doesn't, it will produce an incorrect parse tree that does not include the nested fragment's subtree.

Instead, when the DAC lexer separates a nested fragment from its parent, it substitutes a reference symbol. Reference symbols, from the perspective of the parser, are just normal symbols — new terminals that we have introduced into the grammar. Each reference symbol is mechanically introduced into the grammar; whenever a nonterminal that generates fragments appears in the right-hand side of a parse rule, it is replaced by its own distinct reference symbol. Because the reference symbols are terminals, the parser can complete its task without ever consulting the nested fragments they replace. In this way the grammar is partitioned into two or more subgrammars, one for each nonterminal that generates a type of fragment. The original start symbol can be regarded as creating a subgrammar in the same way; its only distinction is that it happens to generate the root fragment.

A reference symbol also includes as metadata a "subtree pointer" that points to the root of the parse tree of the subfragment it replaces. This parse tree may not yet be constructed, which can be indicated by a special value stored at the location the subtree pointer indicates. When the subfragment is parsed, the subtree's root node is stored at the location indicated by the subtree pointer, so it will always become valid by the time parsing is complete. The DAC parser knows where to store the root node because it is a location agreed upon in advance — for example, space may be reserved at the beginning of each fragment to hold the root node. In this way, the parse trees of all fragments are automatically and implicitly joined to form the complete parse tree without the need for any separate pass to combine them.

[7] An unrolled linked list is one possibility.

In the case of JavaScript, we have only one type of fragment: the function. If functions are generated by the nonterminal `Function` in the grammar, as in Listing 4, then we may introduce the reference symbol `functionref` by transforming the grammar as shown in Listing 5.

```
Program ::= Expressions | Statements
...
Expression ::= MathExpression | LogicExpression
     | Function
Statement ::= LoopStatement |
    ConditionalStatement | Function
Function ::= function Arguments { Body }
...
```

Listing 4: A portion of a JavaScript grammar.

```
Program ::= Expressions | Statements
...
Expression ::= MathExpression | LogicExpression
     | functionref
Statement ::= LoopStatement |
    ConditionalStatement | functionref

Function ::= function Arguments { Body }
...
```

Listing 5: After reference symbols are introduced.

After the DAC lexer processes the JavaScript code in Listing 3, its representation changes from a string of characters to a group of fragments, each containing a sequence of symbols. The original nested structure of the code is expressed implicitly by the reference symbols which the lexer has introduced. If the lexer keeps the symbols for fragment continuous using fragment stack buffers, the resulting data structure will resemble Figure 4.

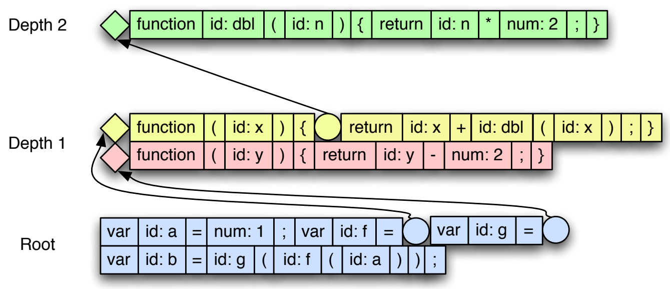


Figure 4: Output of the DAC lexer. Circles represent reference symbols, diamonds represent placeholders for parse tree roots, and the arrows connecting them represent subtree pointers..

3.4 Divide-and-Conquer Parsing

Divide-and-conquer parsing is a system that includes both lexing and parsing components. Most of the changes from standard practice are found in the lexer; modifying a parser to work with the DAC lexer is not much harder than integrating it with any other lexer. This is an advantage of this approach to out-of-order parsing; once the input can be divided into fragments, any algorithm that can handle the grammar can be used to parse the fragments. In this report we focus on LR parsers, but LL parsers or even ad-hoc handwritten parsers can be used.

One of the few restrictions is that the parser must be able to store the root node of the parse tree it generates in a location chosen by the lexer. Even if a parser does not natively support this feature, though, a wrapper function can usually correct the problem.

Some fragments produced by the DAC lexer may need to be parsed with a different start symbol than the root fragment requires. Sometimes the independently parseable units of the grammar can use the same start symbol that the original grammar used, which is very convenient. JavaScript has this property if fragments contain only function bodies, because the statements inside a function can be parsed in the same way as the statements at the outermost level of the JavaScript code. If the fragments are selected in such a way that this won't work, the simplest approach is to generate one parser[8] for each needed start symbol. The parser itself can also be modified to allow the use of different start symbols, which may improve performance by putting less pressure on the CPU cache. This approach abandons the benefit of using an unmodified off-the-shelf parser, though, and it is unlikely to produce any performance gains unless there are many different types of fragments.

The evaluation strategy used by the parser is dictated largely by code in the larger application. Deferred evaluation, for example, is implemented by applying the parser to every fragment produced by the lexer after lexing is complete. Lazy evaluation requires that the application call the parser only when it needs the contents of a particular fragment. Eager evaluation requires only that the lexer call the parser directly whenever it has work for the parser to do.

The semantic actions called by the parser to construct the parse tree may require a little more customization than the parser itself in the face of changing evaluation strategies, though. In particular, with the eager strategy, semantic actions can assume that any subfragments have been parsed, since a child fragment will always finish lexing before its parent, and can inspect their parse trees. With the other strategies, fragments may be parsed in any order, so semantic actions must be written with the possibility that the parse trees of subfragments may not be accessible.

Divide-and-conquer parsing is really just a foundation. It's a generic mechanism for parsing out-of-order that opens up many parsing evaluation strategies that aren't possible with in-order parsing. It's the evaluation strategies that can provide the real benefit, though, by improving parsing performance or by reducing the amount of work that the parser has to do. The main contribution of this paper is a parser that takes advantage of multiple hardware threads to speed up parsing; it's presented in section 4. We explore some of the other benefits DAC parsing can provide in section 6.

4. Parallel Parsing

A divide-and-conquer parser can separate its input into fragments that can be parsed in any order. Once this foundation is available, it's not hard to build a parallel parser on top of it: divide the fragments between the available hardware threads and parse them concurrently. As with most projects involving parallelism, however, getting good speedup is more difficult than it seems.

In this section we present a design for a parallel divide-and-conquer parser. We begin by briefly reviewing our experiences with the most obvious approaches in section 4.1. Most of the work of implementing a parallel DAC parser takes place in the lexer; we present the basic design in section 4.2. Reference tables, which allow each thread to combine its work into the final parse tree, are presented in section 4.3. Finally, issues related to the parser and semantic actions are discussed in section 4.4.

4.1 Parser-Only Parallelism

The parallel divide-and-conquer parser design presented in this report employs a multithreaded lexer instead of the more obvious

[8] Or one copy of the parser data structures, for data-driven parsers.

approach of using a single-threaded lexer to create fragments that can be parsed in parallel in other threads. The problem with the obvious approach is that generating fragments sequentially fast enough to keep several hardware threads constantly supplied with useful work is very difficult. The result is that all of the variations of this approach we tried delivered poor performance.

4.1.1 Static Load Balancing

After the DAC lexer completes execution, it returns a list of fragments. These fragments are then distributed evenly among the available hardware threads. This is the simplest strategy from an implementation perspective, but it fails to produce useful speedup. However, if we ignored the time it took the lexer to run, and compared the time required to parse the fragments in one thread versus that required for four threads, we observed a speedup of about 2x for a 4K test file, and around 3.4x for our test files over 100K. This approach *does* produce a great deal of parallelism, then — it's just that none of the other threads get anything done while the main thread is running the lexer. Since lexing is much faster than parsing, this might be a viable approach with large enough inputs, but it wasn't effective at the input sizes we target in this report.

4.1.2 Dynamic Load Balancing

We implemented a revised DAC lexer using Apple's Grand Central Dispatch library, which provides a task queue abstraction for managing thread pools. Submitting a task is very cheap with this library, requiring only 15 CPU instructions[1]; given this figure, it seemed reasonable to create a task for each fragment as soon as the lexer finished creating it. Unfortunately, the overall overhead involved is such that we observed no speedup, despite fully utilizing all cores in our test machine. Larger-grained tasks didn't help; the synchronization between threads required by the task queue made this approach unworkable for the input sizes we are interested in.

4.1.3 Speculative Load Balancing

To eliminate any requirement for synchronization, we implemented a DAC lexer that simulated static load balancing without waiting for the input to be lexed completely. It assumed that the total number of symbols an input file will generate is a fixed fraction of the input's length; it could then keep track of the number of symbols actually generated and spawn a thread as soon as a fair share of work becomes available. It retains the last share of parsing work for itself.

Given a reasonable choice of ratio between symbols and input length,[9] this method yields a speedup of as much as 2.3x for the input sizes we tested. This is a great improvement over the previous methods, but it still exploits only a little over half of the thread-level parallelism available on our quad-core test machine. The limited speedup is chiefly caused by the significant average delay until a parsing thread starts; on a quad-core machine at least one thread will be idle until 3/4 of the input has been lexed. Unfortunately, subdividing the work further so that tasks were available more frequently did more harm than good in our tests.

4.2 Parallel DAC Lexing

Our parallel DAC parser design obtains concurrency from both parsing and lexing. The input is split into blocks, distributed evenly among the available hardware threads, and a separate lexer runs on each thread. Each lexer locates every independent fragment it can find in its block, lexes it, and parses it while the symbols are still in the CPU cache.

[9] We found that 0.35 was effective for our tests.

Each lexer is assigned a block of the input. The block that starts at the beginning of the input is lexed using the same approach as a conventional DAC lexer with an eager evaluation strategy. The other lexers have to work differently because they are confronted with the problem of missing initial state that we describe for parsers in section 2.3. However, because lexers are so much simpler than parsers, and nesting is not an issue because symbols cannot contain symbols, the problem is less severe. The lexer uses essentially the same strategy as a conservative out-of-order parser (section 2.3.3) — it searches forward through the input, locating fragments that it believes can be lexed and parsed independently.

In the case of JavaScript, the lexer searches for the string "function" in its section of the input, which it speculates corresponds to the JavaScript keyword `function`. It attempts to lex the input starting at that location in the same way that a normal DAC lexer would, parsing the resulting symbols immediately if it finds the right brace that should terminate the function. If parsing succeeds, the lexer continues searching for "function" starting from just after the terminating right brace. If lexing or parsing fails, however, the lexer can establish little other than that it could not recognize a function at that particular point in the input; in this case, it resumes searching immediately after the "function" string that led it astray.

In this way, each lexer will lex and parse every independently parseable fragment that lies within its block. Unfortunately, a JavaScript lexer runs the risk of being mislead into parsing functions that do not really exist. If a comment spans a block boundary and the portion in the later block contains a valid JavaScript function, there'll be no way to distinguish the commented function from one that is actually used. The same problem can occur with functions embedded in strings or regexes. Even though this is possible, however, it's unlikely to be a problem in practice. Functions embedded in strings and regexes are rare, and commented-out functions, while not unlikely, are a low risk in production JavaScript code due to widespread use of minification (discussed in more detail in section 5).

4.3 Reference Tables

As a lexer parses fragments, it records the subtree pointers of top-level fragments in its "reference table" so that other lexers can access them. In the same way that a reference symbol connects the parse trees of different fragments, a reference table connects the parse trees of fragments originating in different blocks. A top-level fragment is either the root fragment or a fragment which is not embedded in another fragment that starts in the same block; these are the only fragments that are needed to reconstruct the entire parse tree, since they contain reference symbols that connect them to all the fragments nested inside them. Each entry in the reference table contains three values: the subtree pointer for the corresponding fragment, the index in the input where the fragment starts, and the index where the fragment ends.

After a lexer records information in its reference table, it never looks at it again; the information in the table is exclusively for the benefit of other lexers. Every lexer begins execution with pointers to the reference tables of the other lexers. When a lexer encounters the end of a block while lexing a potential fragment, it begins to read from the reference table of the lexer which owns the new block it has entered. Every time the lexer encounters a subfragment in the new block, it advances to the next entry in the reference table and checks it to obtain the subfragment's parse tree and ending index. This allows the lexer to insert a reference symbol for that subfragment into the fragment it's working on and skip over it without repeating any work. Since the subfragment can be large, and may itself contain many subfragments, this can allow the lexer to jump forward quite far in the input.

The reference table records potential fragments that parsed successfully, which is a superset of the fragments that a normal lexer would see. In a scenario like the JavaScript case where a function is embedded in a comment that spans a block boundary, as discussed in 4.2, the reference table may contain a superfluous entry. A lexer coming from a previous block and reading that reference table is likely to have seen the beginning of the comment and ignored the embedded function. When it accesses the superfluous reference table entry, then, it will be to obtain the subtree pointer for some function located later in the block. Checking the starting index allows it to detect this situation and move on to later entries in the table, ensuring that the reference symbols the lexer creates refer to the right parse trees.

Reference tables are not protected by any kind of lock. They are write-once data structures that do not have to support concurrent updates, and every value in an entry is word-sized, so that reads and writes of values are atomic.[10] All that is required is to detect if a value has already been written or not. To allow this, reference tables are zeroed out before parsing starts. Reading any value is accomplished by busy-waiting, retrying over and over until the value's contents are no longer zero[11]. The implementation is structured so that reading a value from a reference table entry is a blocking operation; this simplifies the design of the rest of the lexer.

Reference tables have a fixed size, since if a thread resized its reference table it could expose other threads to intermediate states or unpredictable copying delays. To accomodate large numbers of top-level fragments safely, reference tables have a reserved entry that indicates where the next part of that reference table is stored, much like a linked list. Other lexers know when a reference table has run out of space because all reference tables are the same size; when they've read enough entries, they can jump to the next part of the reference table by reading the reserved entry in the same blocking fashion that's used for reading any reference table value.

Reference tables provide two services to a lexer reading their entries; they give it access to parse trees created by other threads, and they let it skip over large parts of the input. Though other threads make use of them, these services are employed the most by the lexer handling the root fragment, which must always traverse the entire input. Few other fragments cross block boundaries; for example, in our `ejdesktop` test input (see section 5), which is one of the largest at 1.3 MB, only 5 of 3948 fragments did so. The root fragment always spans every block, but because most content is in its child fragments, the lexer handling it is able to skip over most of the input from each block.

To understand the association between reference table entries, fragments, and blocks, it may help to have a concrete example. Figure 5 demonstrates how entries are added as lexers progress through a small JavaScript program which has been divided into three blocks.

In Figure 5a, no lexer has encountered any fragments yet, and all reference tables are empty. In Figure 5b, each lexer has reached the end of its own block. No reference table entries are yet filled in. The first lexer has finished parsing function f, but because the first block contains the root fragment, f is not top-level and does not get stored in the reference table.

[10] At least on recent x86 architectures.

[11] To implement this in C, reference tables must be accessed through `volatile` pointers

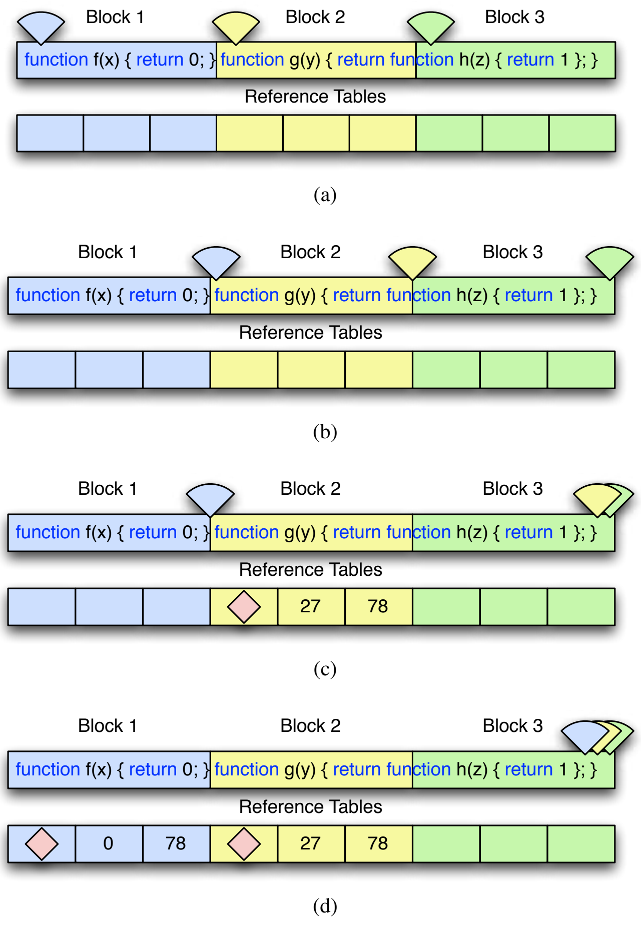

Block 1
Block 2
Block 3
function f(x) { return 0; }function g(y) { return function h(z) { return 1 }; }
Reference Tables
27
78
0
78
(a)
(b)
(c)
(d)

Figure 5: Blocks and reference tables. Wedges represent lexers; diamonds represent subtree pointers.

In Figure 5c, the second lexer has reached the end of the input. It has fully parsed function g, which began in its own block, so Block 2 now has a reference table entry filled in. The entry is associated with the block where the function starts, where the first lexer will read it once it becomes available, allowing it to jump to the end of the input. The second lexer also parsed function h; again, however, it isn't top-level.

Finally, in Figure 5d all three lexers have made it to the end of the input. Only now does Block 1 get an entry in its reference table; this entry is for the root fragment, which spans the whole input. Block 3 never gets an entry because no function starts in that block.

The larger program which employs the parallel DAC parser can get access to the full parse tree by reading the root fragment entry from Block 1's reference table. In a production implementation, a simple wrapper function could be used to make the parse tree available without exposing the internal details of the parallel DAC parser's implementation.

4.4 Parallel DAC Parsing

One advantage of the scheme presented here is that we keep the complexity in the lexer. This is advantageous from a complexity perspective because lexers are simpler than parsers, and it allows us the same flexibility to use almost any parser that we have with a normal DAC lexer. This means that there is little to say about the parsing process itself in a parallel DAC parser; the parsing algorithm doesn't need to be aware of concurrency at all.

Indeed, the semantic actions don't even need to be aware of concurrency unless they touch global state. Each thread uses an eager evaluation order, in which every fragment is parsed as soon as it's lexed. Furthermore, a fragment is never completed and parsed until its children have finished being parsed. This means that parse trees produced by child fragments are available for use in the semantic actions of the parent fragment. With the other approaches to parallel DAC parsing described in section 4.1, this could not be guaranteed, because although a parent fragment would always be constructed after its children, there would be no reference table-like mechanism preventing the parent from actually being parsed first.

This section has presented a design for a parallel divide-and-conquer parser, but it hasn't made any claims as to its performance. We tackle that question in section 5.

5. Evaluation

The parallel divide-and-conquer parser is intended to improve JavaScript parsing performance, resulting in a better web browsing experience for users. In this section, we investigate how well it has achieved this goal by comparing its performance with that of three open source parsers:

ANTLR[28] 3.3 is a widely used lexer and parser generator, intended to have a flexibility and ease of use that makes it suitable for rapid development and teaching. ANTLR generates an LL(*) parser, which is an infinite lookahead version of the more widely known LL(k) top-down parsing algorithm. ANTLR the parser used in the open source JavaScript interpreter Jint[2]. ANTLR can generate code in many languages, including Java, C#, and C++; however, an ANTLR grammar definition generally includes semantic actions written in a particular language, which means that most ANTLR grammars are not language agnostic. To evaluate ANTLR we use a BSD-licensed JavaScript grammar[3] which compiles into Java code; we execute the code using OpenJDK[4] 1.6.0's client VM.

Nitro[5] is the JavaScript engine used in recent versions of WebKit[6]. The version we evaluate in this report is from WebKit 533.19; it uses a parser generated by *GNU Bison*[7] 2.4.1, paired with a lexer generated by *Flex*[8] 2.5.35. Bison can generate either a LALR parser, a member of the LR family of bottom-up parsing algorithms, or a GLR parser, which generalizes LR parsing to allow nondeterminism and ambiguity. The grammar used in Nitro is LALR; our implementation uses a modified version of the same grammar, although we naturally do not use the same lexer or parser.

SpiderMonkey[9] is Mozilla's JavaScript engine, used in Firefox and many other applications. SpiderMonkey uses a custom, ad-hoc JavaScript lexer and parser written in C++ instead of relying on a parser generator. We evaluate version 1.8.5.

All software was executed on a 3 GHz quad-core AMD Phenom II with 512 KB L2 cache per core, running Ubuntu Linux 10.10. Experiments that measurement time are always run 11 times for each input, with the first run ignored; the reported value is the arithmetic mean of the remaining runs.

We present our test inputs in Table 1. `hello` is a simple JavaScript program with a single function that prints "Hello world!" and exits; it's the only input created just for this report. `dojomob`, `jquery`, and `prototype` are widely-used real world libraries for creating rich web applications with JavaScript. All are structured as a base library which can be combined with various modules or plugins; our test input consists only of the base library. `yui` and `closure` are similar libraries created by Yahoo! and Google respectively. They differ from the other libraries in that there is no canonical base library; all code is either loaded dynamically or "baked" into a custom library containing only what is needed for a particular application. Our test inputs consist of all modules from the standard distributions of these libraries; this approach is somewhat unrealistic, as few applications today are likely to use all the modules these libraries provide, but as web applications increase in complexity code of this size is likely to become more prevalent. `box2d` is a physics library frequently used for implementing JavaScript simulations and games. `jqcal` and `ejdesktop` are demo web applications build using the jQuery and ExtJS frameworks respectively; they are representative of the kinds of applications these frameworks are made to facilitate.

"Minification" is a common technique in JavaScript development which reduces bandwidth costs, improves page load time, and obfuscates code to protect the intellectual property of its creators. All whitespace and inline comments are removed from the source code, and any local identifiers are replaced with generated names that use as few characters as possible. Since this technique is standard in the real world, we used minified versions of inputs whenever a choice was available; all of the libraries listed above except for `box2d` are minified. `hello`, `jqcal`, and `ejdesktop` are not minified.

5.1 Results

We summarize the results of our experiments in Table 2. Parsing times for each input are compared for each of the parsers in Figure 6. Because the times range from tens of microseconds to seconds, the graph is presented in a log scale. We depict the speedup achieved by the parallel divide-and-conquer parser on each input in Figure 7.

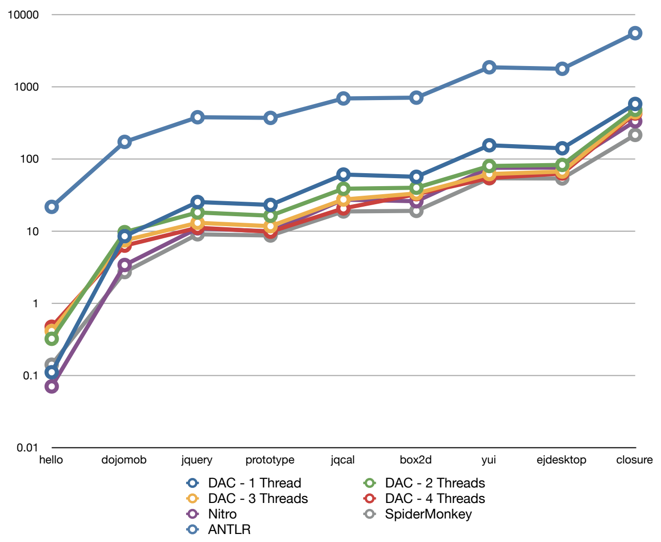


Figure 6: Parsing times for all parsers and inputs, in milliseconds. Log scale.

ANTLR is clearly the poorest performer. Its performance problems are not exclusive to Java code; though we do not present the results here, our experience with the Jint JavaScript interpreter, which uses an ANTLR gramar that targets .NET, has been quite similar. Though we do not investigate the details here, its high memory usage is suggestive — while none of the other parsers ever even

Input	Name	Size	Description
`hello`	Hello World	82 B	Simple “Hello World” JavaScript program
`dojomob`	Dojo Mobile[10] 1.6	35 kB	Mobile JavaScript GUI toolkit
`jquery`	jQuery[11] 1.6	89 kB	Popular JavaScript library
`prototype`	Prototype[12] 1.7	160 kB	JavaScript web application framework
`jqcal`	jQuery Week Calendar[13] Demo 1.2.3-pre	298 kB	Calendar management web application demo, based on jQuery 1.3.2
`box2d`	Box2D.js[14] 0.1.0	381 kB	JavaScript physics engine
`yui`	YUI[15] 3.3.0	736 kB	Yahoo!’s JavaScript web application library
`ejdesktop`	ExtJS Desktop[16] Demo	1.3 MB	Demo simulating a desktop user interface, based on ExtJS 3.2.1
`closure`	Closure[17] r903	4.0 MB	Google’s JavaScript web application library

Table 1: Test inputs for evaluating parser performance, arranged in order of size.

	Time (milliseconds)								
Parser	`hello`	`dojomob`	`jquery`	`prototype`	`jqcal`	`box2d`	`yui`	`ejdesktop`	`closure`
ANTLR	21.54	171.21	375.87	368.08	684.28	702.09	1848.73	1762.34	5498.5
Nitro	0.07	3.37	10.84	9.86	27.12	25.77	74.85 ms	75.34	330.51
SpiderMonkey	0.14ms	2.69	8.97	8.64	18.61	18.95	53.59	53.28	213.86
PDAC (1)	0.11	8.45	25.13	22.92	60.40	56.22	153.66	139.54	574.16
PDAC (2)	0.32 (0.34x)	9.61 (0.88x)	18.02 (1.39x)	16.24 (1.41x)	38.41 (1.57x)	39.64 (1.42x)	79.28 (1.94x)	82.10 (1.70x)	476.15 (1.21x)
PDAC (3)	0.41 (0.27x)	7.38 (1.14x)	12.95 (1.94x)	11.63 (1.97x)	27.22 (2.22x)	33.04 (1.70x)	61.34 (2.51x)	66.44 (2.10x)	440.14 (1.30x)
PDAC (4)	0.47 (0.23x)	6.25 (1.35x)	11.06 (2.27x)	9.75 (2.35x)	20.51 (2.94x)	32.26 (1.74x)	54.01 (2.85x)	62.44 (2.23x)	422.94 (1.36x)

Table 2: Summary of experimental results. For the parallel DAC parser, the number of threads used in each trial is indicated in parentheses.

reach 100 MB of peak memory usage, ANTLR crosses that threshold even for `jqcal`, an input less than 300 K in size, and it exceeds 500 MB for `closure`. Making such large allocations for such small problem sizes is bound to affect performance negatively.

Nitro and SpiderMonkey are both fast, with SpiderMonkey having a 25 – 30% edge on larger inputs. SpiderMonkey has a custom, hand-tuned parser, which likely accounts for some of the difference. Nitro’s parser is data-driven, while in SpiderMonkey the parser is implemented directly in code; this design choice alone can make a significant difference in performance. Another contributing factor is the implementation of semicolon insertion, a feature of JavaScript that allows programmers to omit semicolons that are required by the language’s grammar. Nitro implements semicolon insertion within the limitations of Bison partially by detecting parse errors that could have been caused by a missing semicolon and retrying recent parsing steps as if a phantom semicolon has been inserted into the input. SpiderMonkey’s ad-hoc parser detects situations where a semicolon must be inserted before any problem occurs, so it never has to retry. Since even JavaScript programmers who use semicolons frequently leave them out for aesthetic purposes in many places (for example, after a function body), less efficient handling of semicolon insertion can really add up over the span of a large JavaScript file. Unfortunately, because the parallel DAC parser uses essentially the same grammar as Nitro relies on a similar parser design, it shares the same issues.

The parallel DAC parser has the weakest sequential performance of any of the parsers examined other than ANTLR. However, this limitation is only in small part due to the overhead of the divide-and-conquer algorithm or support for parallelism. In the single-threaded case, the main thread invokes a single lexer on the entire input; the overhead over a normal lexer consists of the allocation of unnecessary pointers and indices used to access the reference buffers of other threads, at a cost of four words of memory, and the management of the fragment stack, which involves incrementing and decrement pointers, with dynamic allocation only occurring in the quite rare case that the fragment stack needs to be expanded. Though the lexer could surely benefit from further optimization, it

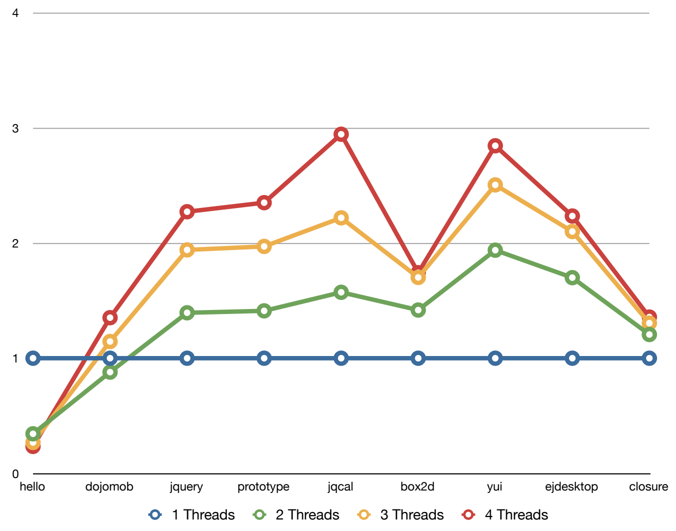


Figure 7: Parallel DAC parser speedup over the single-threaded case.

isn’t likely to be the primary source of the disparity between the parallel DAC parser and the more mature parsers.

The parallel DAC parser uses a generic, data-driven LALR parser design which was not created specifically for JavaScript. Beyond the concerns mentioned for Nitro above, the details of its semicolon insertion implementation require that an entire fragment be buffered before it can be parsed, which is even worse than Nitro. Fortunately, nothing in the parallel DAC parser’s design requires the use of a particular parsing algorithm, and though modifying the SpiderMonkey parser so that it could be called cleanly by a DAC lexer would be challenging, it is certainly possible technically.

The speedup the parallel DAC parser offers, then, is the most important criteria on which it should be judged, since the parsing algorithm used internally can be replaced without affecting how work is distributed between threads. As Figure 7 shows, the parallel DAC parser is not capable of obtaining speedup on `hello`. That input is very small and contains only one function; since we obtain parallelism from finding and parsing functions in parallel in each thread, `hello` offers no work for additional threads to do. However, `dojomob`, despite being only 35 kB, is large enough to get speedup from parallelism, and the trend continues with larger inputs.

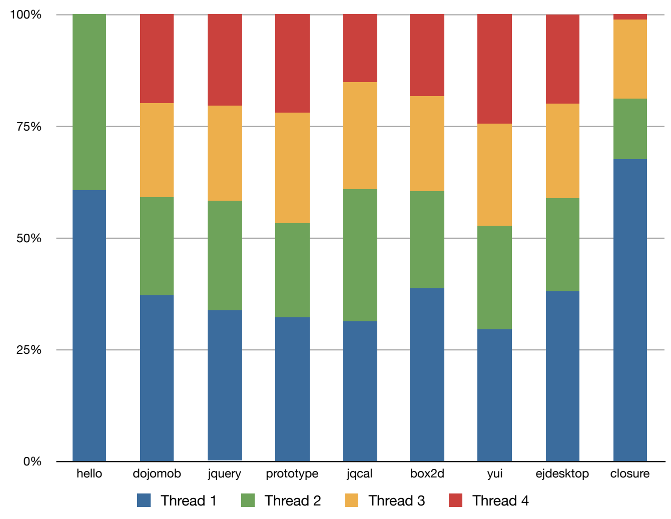


Figure 8: Load balancing behavior of the parallel DAC parser, running with 4 threads, measured as the percentage of symbols each thread lexed and parsed.

The speedup doesn't scale ideally — that is, doubling the number of threads doesn't double the speedup. In fact, adding an additional thread only improves performance on most inputs by 0.4 – 0.5x. To investigate this behavior, we instrumented the parallel DAC parser to report how many symbols each thread lexed and parsed. The results, shown in Figure 8, show a clear correlation with speedup: the more work the first thread performs relative to the others, the less speedup we see. Recall that the first thread is responsible for the root fragment, and must lex and parse any portion of the input that is not wrapped in a function. This fact allows us to explain the results in terms of the inputs: those inputs with the least speedup are uniformly those with the most content in the root fragment. The two inputs which stand out as anomalies with especially bad performance, `box2d` and `closure`, both consist of a long list of functions defined at the top level, with none of the hierarchical structure typical of the other JavaScript libraries.

The speedup penalty that `box2d` experiences seems out of proportion to its work imbalance, however. There's a second effect that contributes to the problem: lexing is much faster than parsing, and since the current implementation of the parallel DAC lexer does not perform parsing until a fragment is completely lexed, if the first thread finishes with its block before the other threads are done, it will only lex from that point until the end of the input. This gives it a good chance of catching up to the other threads. This happens with most of the small-to-mid-size inputs, as depicted in Figure 9, which shows the number of retries each thread required when attempting to read the reference table entries of functions outside its block. Such retries occur when the thread responsible for parsing the function has not yet finished doing so; since they occur in a busy-wait loop, a huge number of retries can be generated each time this happens.`box2d`'s first thread required many more retries than any other input experienced, indicating that it caught up to the other threads very fast.

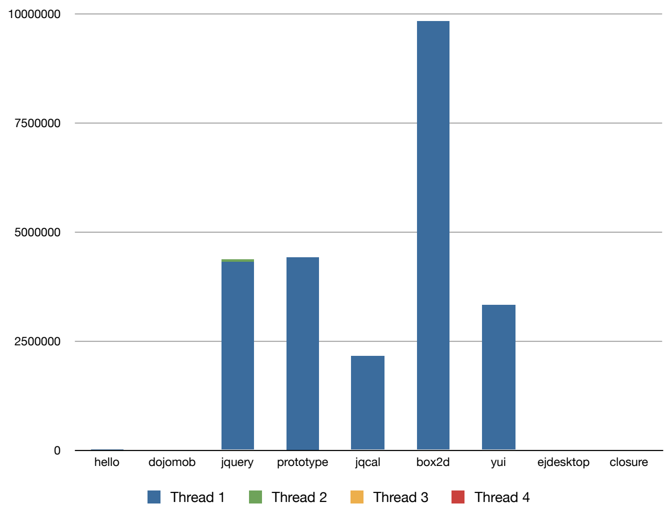


Figure 9: Number of reference table read retries required by the parallel DAC parser, running with 4 threads.

Though the structure of `box2d` triggers this issue particularly badly, all of the smaller inputs are affected. Fortunately, it is likely that switching to a parsing algorithm that can handle semicolon insertion without requiring access to previous symbols would go a long way to alleviate this problem; with this modification, the first thread could parse the root fragment incrementally as it lexes it, slowing it down significantly and reducing retries greatly. Right now parsing the root fragment cannot be overlapped with any other work, because every other fragment is directly or indirectly embedded in the root fragment. Being able to parse fragments incrementally would eliminate this issue as well.

6. Extensions

Divide-and-conquer parsing is useful for more than just the parallel parsing algorithm discussed in section 4. In this section we discuss some other techniques that become feasible once we eliminate the restriction that parsing must happen in-order.

Section 6.1 describes lazy parsing, a parsing evaluation strategy which improves performance by avoiding unnecessary work. Lazy parsing is normally incompatible with parallel parsing; we discuss one way of address this limitation in section 6.2.

6.1 Lazy Parsing

A divide-and-conquer lexer produces a set of fragments that can be parsed independently, giving us a great deal of flexibility to choose when each fragment should be parsed. We have so far implicitly assumed that all of these fragments are immediately useful, and so both the basic divide-and-conquer parser and the parallel version have used an "eager" parsing strategy: though the exact timing may vary, every fragment that is produced is parsed.

However, in the real world many fragments do not need to be parsed immediately. Fragments are not useful until the main application requires the data they contain to make computational progress, which may not be for a long time in many cases. Indeed, many fragments may never actually be referenced, and so parsing them at all would be a waste of time.

"Lazy" parsing, a straightforward extension to the basic divide-and-conquer parser, provides a solution to this problem. The lexer returns all the fragments it produces immediately, unparsed. The larger application then accesses the parse tree as usual, but it must now check if each pointer to a fragment's subtree is null before dereferencing it. If so, the application invokes the parser to parse that fragment, which creates the missing subtree and updates the subtree pointer for use in subsequent accesses. The application can then access the subtree through the pointer and continue its work. This process is illustrated in Figure 10.

For lazy parsing to work as intended, parsing the root fragment cannot implicitly cause all other fragments to be parsed — otherwise, as soon as the application attempted any access to the parse tree, the entire tree would be created, and there would be little benefit to the technique. More generally, a fragment's semantic actions should not access the parse trees of its child fragments unless it is certain that those child fragments are about to be needed. To make this concrete, let us return to the example of JavaScript, with fragments corresponding to function definitions. It would be inappropriate, in a lazy DAC parser, for semantic actions to trigger the parsing of a function definition merely because it is encountered in source code. Instead, functions should only be parsed when they are actually needed — for example, when an executing script calls them. A JavaScript engine making use of a lazy DAC parser would thus begin by parsing and executing the root fragment, parsing additional fragments as required by dynamically-encountered function calls.

For some languages, it may be beneficial to make the lexing process lazy as well. With this approach, only one fragment is actually lexed at a time; child fragments remain unlexed until they are needed for computation to progress. Of course, lazy lexing cannot really be *totally* lazy, as at a minimum it's still necessary to scan through a child fragment to locate its boundary. If most of the normal activity of the lexer is unnecessary for this simplified problem, it's possible that lazy lexing could provide a performance boost. In the case of JavaScript function definitions, unfortunately, a lazy lexer still retains a fair degree of complexity. For example, the problematic character '/' may indicate the beginning of a regular expression, inside which brace characters would not affect nesting depth, or it may merely indicate division; to distinguish between the two cases, it's necessary to correctly recognize a number of other symbols as well, because the correct interpretation depends on context. The details of this problem are related to a particular quirk of lexing JavaScript and a particular fragmentation approach, but we suspect that similar problems will be encountered in many other languages of real-world interest, rendering lazy lexing too close in complexity to normal lexing to provide any substantial benefit. We were not able to realize any speedup from lazy lexing in preliminary experiments.

Semantic actions may have to be written quite differently, depending on their requirements, for a lazy DAC parser than for an eager DAC parser or the parallel DAC parser presented in this report. In the latter cases, a semantic action may assume that all child fragments of the current fragment have already been parsed, and can access their parse trees freely; however, parent fragments have certainly not been parsed. In the lazy case, exactly the opposite is true: parent fragments will always be parsed before their children, while the parse trees of child fragments won't be available unless semantic actions force parsing to occur. Thus, lazy DAC parsers execute their semantic actions top-down, while the other DAC parsers we've discussed execute their semantic actions bottom-up; this may be a benefit or a drawback, depending on how data needs to flow within a particular application, and so it may be a factor when choosing between the parallel and lazy approaches.

Given that we intend the methods of this report to ultimately improve the performance of real-world web applications, it's worth asking how much benefit we can obtain from using a lazy DAC parser instead of a standard JavaScript parser in a browser. A study of popular web applications concluded that only 50 – 65% of functions they defined were actually called[29]; the rest were either dead code or were not invoked except under particular circumstances. Assuming that the average size of dead and live functions is about the same, this suggests that we can expect a lazy DAC parser to perform as little as half the work of a standard parser, yielding a 2x speedup, although some of that benefit will be lost due to the slightly increased complexity of a DAC lexer.

However, perhaps more important for user perception of browser performance than overall speedup are startup costs. Since the initial rendering of a web page blocks when JavaScript code is encountered and cannot continue until the code finishes executing, minimizing the amount of time that process takes is very important to improving page load time. Here laziness can be even more beneficial, since in many cases only a minority of functions are executed during initialization. Many libraries, in particular, do little work at initialization time other than storing functions into a JavaScript hash table. In such a case, very little code needs to be parsed at initialization time, and we expect to eliminate nearly all parsing work. In experiments we conducted on lazily parsing major JavaScript frameworks like JQuery and Prototype, we saw initial parsing time improvements of 95 – 98%. The functions that are actually called will eventually need to be parsed, but to the extent that that work is delayed until the page is fully loaded and responsive to input events, the user experience will be improved.

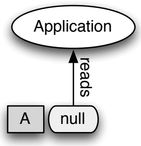
Lazy divide-and-conquer parsing is a promising technique for reducing page load time in browsers and eliminating unnecessary work that reduces performance and increases energy usage. Our preliminary results suggest that the technique is practical and effective.

6.2 Speculative Parsing

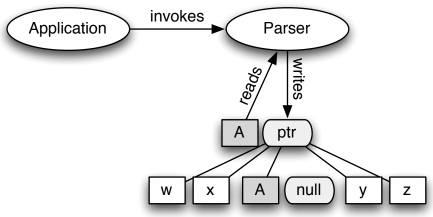
Lazy divide-and-conquer parsing generally only parses one fragment at a time, because a fragment isn't parsed until it's needed in the context of a larger computation that's usually sequential. The result is that lazy parsing normally does not offer any opportunities for parallelism. To improve performance beyond what laziness can offer, it would be useful to combine laziness and parallelism in the same parser. We propose speculative parsing as a method for achieving this goal.

While a fragment is being parsed, a speculative parser performs additional analysis to try to infer other fragments that are likely to be needed in the near future. Any fragments that it identifies are parsed in one or more threads separate from the main computation. Speculatively parsed fragments which are actually needed later will be available to the application as part of the parse tree, just as if they had been parsed in the normal fashion.

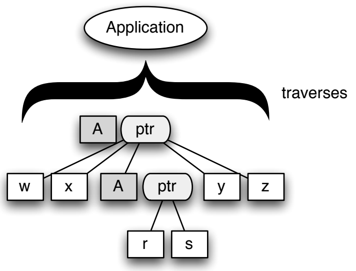
There are many heuristics a speculative parser could use to identify fragments worth parsing. The simplest approach would be to speculatively parse every fragment the parser encounters, so that free processor time is used to gradually parse the whole input, while the lazy parser ensures that whatever fragment is most urgently needed is parsed immediately if it's not already available. More sophisticated heuristics may be used to maintain as much as possible of the reduction in work that lazy parsing enables, or to increase the odds that fragments that are actually needed are parsed first. In general, a speculative parser can give a score to fragments based on various features it observes and parse higher-scoring fragments first, with



(a) The application attempts to access the root fragment's parse tree pointer, which is initially null.



(b) The null pointer tells the application to invoke the parser, which reads the fragment and outputs a parse tree.



(c) As the application accesses subtrees, it invokes the parser whenever it finds a null pointer, gradually generating every useful part of the parse tree.

Figure 10: The lazy parsing process.

fragments scoring under a certain threshold not parsed at all unless the main application requests it.

In the JavaScript case, a parser can observe which functions may be called in the current fragment and speculatively parse fragments which may contain those functions. To reduce false positives, the parser may take into account features such as the presence and nesting level of conditional expressions, or whether the function in question is passed to another function as an argument (to be used as a callback, for example). There may be many functions with the same name, and the parser frequently will not be able to determine which one would be invoked at runtime; thus, the number of functions with the same name may serve as a useful feature to decide whether parsing all of the corresponding fragments is worthwhile. We observed in our tests that the vast majority of function names are unique to one function or a very small number of functions, but there are a small number of function names that are reused hundreds of times due to object oriented programming patterns used in some frameworks; this programming style can make speculative parsing less precise.

We performed a preliminary experiment to evaluate the feasibility of speculative parsing. We ran the JavaScript demo web applications `jqcal` and `ejdesktop` introduced in section 5 and collected execution traces showing which functions were actually called during a realistic user session. We then simulated the behavior of a speculative parser many times with different sets of features in an attempt to determine how effectively such a parser could predict which functions would be called without any runtime information. The best performance we observed was with the JQuery Calendar demo, where we were able to correctly speculatively parse 63% of 268 called functions at a cost of incorrectly parsing 11% of the 880 functions that weren't called in our trace.

More research is needed to identify good feature sets to guide speculative parsing, and to evaluate its performance as part of a running browser with JavaScript execution informing its behavior. Nevertheless, we believe that these initial results hold promise, and that the combination of laziness and parallelism would provide compelling benefits if realized.

7. Related Work

Parsing is a subfield of computer science which has been actively researched for decades, and many approaches to parallel parsing have been proposed over the years. A full review of the parallel parsing literature is beyond the scope of this report; interested readers will find useful surveys of the field in [23] and [25]. We will focus instead on clarifying our contribution to the field by contrasting parallel divide-and-conquer parsing with competing solutions — that is, solutions that operate on the same type of computer architecture and are applicable to the problem of parsing in the browser.

7.1 File-based Parallelism

If multiple files need to be parsed — for example, multiple JavaScript files embedded into a single web page — then file-based parallelism may be a competitive solution. Separate files can be parsed completely in parallel, using traditional sequential techniques, with no interference. A limitation of this approach, though, is that in practice each file often needs to be delivered in sequence over a network that may be so slow that parallel parsing each file individually is still faster; an application can take measurements of network speed into account to decide which granularity of parallelism would be better.

7.2 General Parallel Parsing Techniques

There are several existing general purpose parallel parsing techniques that are substantially different from the techniques presented in this report. Many of them are designed for natural language processing, and rely on the ambiguity rampant in natural languages. They find parallelism by parsing the same input using many different interpretations at once. Computer languages such as JavaScript are generally designed to contain no ambiguity, so we will not consider this class of algorithms further.

Fisher presents a parallel extension of LR parsers called "synchronous parsing machines" (SPMs) which operate by running many LR parsers in parallel on the same input[19]. Because each parser cannot know its initial state, it starts with a set of states guaranteed to contain the correct one, and updates them all; this is a speculative algorithm. Invalid states will generally lead to an error sooner or later and be unable to continue; thus, each parser will be responsible for fewer and fewer states over time, until only the correct one is left. This algorithm is similar to parallel divide-and-conquer parsing in that it makes use of LR parsers which are almost unmodified. However, the amount of speculation it performs leads to a great deal of overhead, and perhaps as a consequence Fisher reports far sublinear speedups; he does not achieve better than 25% efficiency, while our best results are 74% efficient.

Chart parsers have spawned a whole family of parallel parsing techniques. While LR and similar parsers manipulate a stack to parse input, chart parsers fill out a table. For example, for the well-known Cocke-Younger-Kasami chart parsing algorithm CYK[21], the table is conceptually three dimensional, with entry $[i, j, R]$ containing the value true if the region of the input starting at i and ending at j can be generated by rule R in the grammar. Assume that one way this can happen is if rule S generates the input from i to k and rule T generates the input from k to j. This means that if entries $[i, k, S]$ and $[k, j, T]$ are set to true in the table, CYK knows that it can set $[i, j, R]$ to true. In this way, data flows through the table from cells for rules that match individual symbols all the way up to cells that match the whole input. (i.e., those corresponding to the start symbol) Parallelizing chart parsers, then, is a matter of using multiple hardware threads to fill in many cells in the table simultaneously, choosing the order in which the cells are filled to minimize dependencies and hence communication. This can be done effectively in practice, and parallel chart parsers often yield significant speedup[30]. The main tradeoffs are that chart parsers start out significantly slower than the LR family of parsers, so their speedup is less meaningful, and that chart parsers can require a lot of memory, which led some of them to be considered impractical when they were first introduced[20]. We now have sufficient main memory to make chart parsers a viable alternative, but CPU caches are still small, and so the less memory-intensive LR family of parsers is still the better choice in performance-critical applications.

7.3 Parallel XML Parsing

The existing proposals most similar to divide-and-conquer parallel parsing are found in a recent body of research addressing the problem of speeding up XML parsing using laziness and parallelism.

Noga et al. present a lazy XML parser with several initial preprocessing stages that together perform a similar task to our divide-and-conquer lexer.[26] The input is scanned once to convert it to Unicode and a second time to lex it, producing a sequence of symbols that are scanned a third time to construct a tree representing the structure of the document. Each node in the tree stores an index into the symbols produced by the lexer which indicates its contents. The nodes can then be parsed lazily as the larger application requests them.

Lu et al. describe a parallel XML parser with an initial "preparsing" stage that constructs a skeleton of the final XML DOM[24], much like that described in Noga et al. The authors divide the skeleton into subtrees that are parsed in parallel. They use two schemes for doing so: a static scheme that uses a separate pass over the skeleton to partition it into equally sized subtrees, and a dynamic scheme that uses a thread pool with work stealing.

A followup paper by Pan et al. extends the parallel XML parser described above by parallelizing the preparsing stage as well[27]. They replace the DFA used for preparsing with a meta-DFA that has an initial state for every state of the original DFA. They divide the input evenly into blocks and start a meta-DFA at the beginning of each block; the meta-DFA effectively starts in all states of the original DFA at the same time, and eliminates impossible states as it progresses through the input. Each block, after being processed by a meta-DFA, yields a set of trees representing every possible structure of that block. Once all blocks have been processed, the correct tree for each block can be selected and spliced into a final tree representing the structure for the entire input; the tree can then be partitioned between processes, and parallel parsing can begin.

Our approach resembles all three of these schemes, but differs from them in several ways. One fundamental difference is that the natural definition of a fragment in XML is always contiguous, which is not true for many other languages, including JavaScript. Given the XML document:

```
<parent>
        child1
        <child2>
                (more elements...)
        </child2>
        child3
</parent>
```

We can treat `child1` and `child3` as two different fragments which can be parsed separately. If XML had the same kind of grammatical structure as the more complex languages our work is intended to parse, `child1` and `child3` would have to be parsed together — that is, they would constitute a single fragment. For this reason, we could not represent a fragment using a simple index into an array of symbols, as these XML parsers do. We contribute a solution to this problem by using fragment stack buffers, as discussed in section 3.2.

Our approach, being integrated into the lexer, requires no additional preprocessing for lazy or parallel parsing. Indeed, in the parallel divide-and-conquer parser even lexing is parallelized. This contribution is an improvement over the first two XML parsing schemes. The third scheme also parallelizes lexing, but does so in a much more speculative way; our method only attempts to lex and parse starting from what appears to be the beginning of a fragment, giving us a high confidence that our work will be useful, while their approach will start at the beginning of a block regardless of what is found there. The advantage of their method is that they potentially achieve better load balancing, since the amount of work a thread has to do is more directly related to the size of the block it is assigned; our approach, on the other hand, greatly simplifies the construction of the final parse tree by ensuring that the work of each thread has a simple structure, so that we do not need a separate sequential pass to merge the results of each block together.

By not having a separate planning phase before parsing begins, we improve on the approaches above because we can parse each fragment as soon as it is lexed, while the cache is still warm; this is particularly important for mobile devices which may have processors with very small caches. We do not pay the cost of partitioning a skeleton between processors statically, nor do we use work stealing to balance the load between processors, avoiding its overhead.

8. Conclusion

As computer users spend more and more of their time in a web browser, and become increasingly accustomed to web applications that are maintenance free, effortlessly in sync, and work on every platform, commercial pressure will increase to move even traditional native applications onto the web. To deliver the performance and feature set users have come to expect from these applications, though, every component of the web application stack will have to be improved.

We tackle one such component in this report: by building a faster parser, we hope to reduce the delays in web application startup caused by the need to lex, parse, analyze, and compile JavaScript at page load time. The foundation of our approach is divide-and-conquer parsing, which frees us from the need to parse input from left to right. We use this technique to build a parallel divide-and-conquer parser which takes advantage of the multicore processors becoming increasingly prevalent in both traditional PCs and mobile devices.

Parallel divide-and-conquer parsing provides speedup of as much as 2.94x with four threads, and it gracefully degrades to a single-threaded parser with acceptable performance, without any modifications to the algorithm. Even on inputs which are not suitably structured for divide-and-conquer parsing, we still observe speedup, and inputs as small as 35 kB can benefit from parallelism. Though the implementation evaluated in this report suffers from limitations as a result of the underlying parsing algorithm, we believe that these results show that parallel divide-and-conquer parsing has promise.

Divide-and-conquer parsing can also be used to implement lazy parsing, which can improve parsing performance by not parsing content that is not actually used. We project a lazy parser to yield a speedup of about 2x on real-world JavaScript content, and improve page load time even further because much parsing can be deferred until after the page is loaded. Combined with speculation, which allows lazy parsing and parallel parsing to be combined, we believe that this is a promising direction for future research.

To reach their potential as a universal computing platform, browsers will need to provide native application performance even as they transcend native application limitations. Divide-and-conquer parsing, by making it possible to reduce web application startup time, is a small step toward this distant but important goal.

9. Acknowledgements

This report was made possible by the contributions and support of many talented people. The readers, Professors Rastislav Bodík and Eric Brewer, offered invaluable feedback and assistance in the completion of this report and showed remarkable patience during the process. Additionally, most of the work was performed under Professor Bodík's supervision, and his insightful guidance greatly improved the final product. The work on lazy and speculative parsing was done during a summer internship at Qualcomm Inc. under the wise and gracious supervision of Calin Cascaval and Mehrdad Reshadi; they not only contributed their knowledge and experience but also kindly allowed a summary of the work to be included in this report. Leo Meyerovich, Yahel Ben-David, Matthias Vallentin, and Javier Rosa provided feedback, greatly appreciated support, and many an interesting conversation as this work took shape. My partner, Jayde Terry, showed amazing understanding and patience with my long hours, for which I cannot thank her enough. Inevitably there are some people who have been left out of this list, but every contribution of knowledge, support, or assistance was valuable and greatly appreciated.

References

[1] `http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf`.

[2] `http://jint.codeplex.com`.

[3] `http://research.xebic.com/es3`.

[4] `http://openjdk.java.net`.

[5] `http://www.webkit.org/blog/214/introducing-squirrelfish-extreme`.

[6] `http://webkit.org`.

[7] `http://www.gnu.org/software/bison`.

[8] `http://flex.sourceforge.net`.

[9] `http://www.mozilla.org/js/spidermonkey`.

[10] `http://dojotookit.org`.

[11] `http://jquery.com`.

[12] `http://prototypejs.org`.

[13] `https://github.com/robmonie/jquery-week-calendar/wiki`.

[14] `http://box2d-js.sourceforge.net`.

[15] `http://developer.yahoo.com/yui`.

[16] `http://dev.sencha.com/deploy/ext-4.0.0/examples/desktop/desktop.html`.

[17] `http://code.google.com/closure`.

[18] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In Oscar Ibarra and Zhe Dang, editors, *Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin / Heidelberg, 2006.

[19] Charles N. Fischer. On parsing context free languages in parallel environments. Technical report, Ithaca, NY, USA, 1975.

[20] Bryan Ford and M. Frans Kaashoek. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.

[21] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages, 1965.

[22] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.

[23] M. P. Van Lohuizen and M. P. Van Lohuizen. Survey of parallel context-free parsing techniques, 1997.

[24] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to XML parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 223–230, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Anton Nijholt. Overview of parallel parsing strategies. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, chapter 14. Kluwer Academic Publishers, Norwell, MA, 1991.

[26] Markus L. Noga, Steffen Schott, and Welf Löwe. Lazy XML processing. In *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, pages 88–94, New York, NY, USA, 2002. ACM.

[27] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu. Parallel XML parsing using meta-DFAs. In *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 237–244, Washington, DC, USA, 2007. IEEE Computer Society.

[28] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[29] Paruj Ratanaworabhan, Benjamin Livshits, David Simmons, and Benjamin Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2010-8, Microsoft Research, January 2010.

[30] Greg Sandstrom. A parallel extension of Earley's parsing algorithm, 2004.