Journal of Information Processing Vol.23 No.4 505–512 (July 2015)

[DOI: 10.2197/ipsjjip.23.505]

Regular Paper

Packrat Parsing with Elastic Sliding Window

Kimio Kuramitsu[1,a)]

Received: October 14, 2014, Accepted: February 19, 2015

Abstract: Packrat parsing is a linear-time implementation method of recursive descent parsers. The trick is a memoization mechanism, where all parsing results are memorized to avoid redundant parsing in cases of backtracking. An arising problem is extremely huge heap consumption in memoization, resulting in the fact that the cost of memoization is likely to outweigh its benefits. In many cases, developers need to make a difficult choice to abandon packrat parsing despite the possible exponential time parsing. Elastic packrat parsing is developed in order to avoid such a difficult choice. The heap consumption is upper-bounded since memorized results are stored on a sliding window buffer. In addition, the buffer capacity is adjusted by tracing each of nonterminal backtracking activities at runtime. Elastic packrat parsing is implemented in a part of our Nez parser. We demonstrate that the elastic packrat parsing achieves stable and robust performance against a variety of inputs with different backtracking activities.

Keywords: Parsing Expression Grammars, packrat parsing, backtracking activity, parser generators

1. Introduction

Packrat parsing is a popular technique for implementing recursive descent parsers with backtracking. The main idea behind packrat parsing is an incremental memoization that works by storing all intermediate results parsed at each distinct position in the input stream. Since each nonterminal is called no more than once, packrat parsing avoids the potential of exponential costs.

Packrat parsers are widely used with *Parsing Expression Grammars* or PEG-extended grammars. A PEG [4] is a syntactic foundation that can recognize a wide range of formats, not only programming source code but also data formats such as XML and computer logs. As practitioner experiences have increased over such variety, many developers have pointed out that packrat parsing might not fit in its memoization costs, or may be even less efficient than a plain recursive descent parser. One particular reason lies in its huge linear memory consumption, requiring up to 100–400 bytes of memory for every byte of input [3], [5]. In practice, memory consumption of this scale can cause several other performance degradations, such as disk slashing and repeated garbage collector invocations. That is why not a few developers, despite admitting the desirable property of linear time guarantee, have abandoned the benefits of memoization [11].

In our development experience of Nez grammars, or a PEG-extended grammars [6], we observed no or very little backtrack activity in parsing data formats, such as XML and JSON. In such cases, a plain recursive descent parser is extremely superior, while packrat parsing shows poor performance, especially in cases of a large-scale input sizes. In cases of parsing C source code, despite a fact that we can observe significantly increased backtracking activity, a plain parser still shows very competitive performance compared to a packrat parser. The advantage of packrat parsing is still questionable if we take its huge memory consumption into consideration.

As many researchers have claimed [1], [9], [11], exponential behavior does not seem to happen in practice. However, our experience suggested that the claim is not true in some closed visible examples. For example, parsing a common source of JavaScript, such as JQuery library, involves a considerable high-level backtracking. Without any memoization supports, the parsing of JQuery libraries takes a literally exponential time.

The problem is that packrat parsing does not handle well a wide range of input variations, especially very large-scale files and low backtracking activity cases. The purpose of this paper is to make the performance of packrat parsing robust against such input variety in size and underlying backtracking activities.

Our idea behind packrat parsing with elastic sliding window (or simply *elastic packrat parsing*) is based on the observation of the *worstlongest backtrack backtrack length*. In principle, memoized results are just needed within the range of the longest backtrack from the head position (i.e., the maximum matched length at each backtracking time). If a small memorization table (called window) slides and covers the longest backtrack, the space is sufficient enough to keep all results to avoid redundant calls. In practice, however, it is hard to know the longest backtrack before parsing. Alternatively, we select an approximated window size from the empirical investigation and, if necessary, expand it during the parsing process.

We take a constant-space approach to the expansion of the window size. First, we trace the utilization of memoization at each nonterminal call in order to reduce unused memoization space. Then, we prepare a flexible table structure that could allow reduced nonterminal spaces to be available as if the window size were expanded. Due to this structure, we can achieve the graceful coverage of window size limits while keeping constant heap consumption.

[1] Yokohama National University, Yokohama, Kanagawa 240–8501, Japan

[a)] kimio@ynu.ac.jp

Our elastic packrat parsing is implemented in a part of the Nez dynamic parsing library [6]. Although Nez provides some extended operators to transform parsed results into an Abstract Syntax Tree representation, we evaluate the parser performance without any construction of ASTs. The proposed algorithms and experimental results can be applied to enhance any existing packrat parsing.

Our experimental study demonstrates very promising results. In lower backtracking activity cases, although memory pressure causes considerable slowdown in conventional packrat parsing, our elastic packrat parser shows a competitive performance compared with a plain recursive descent parser. In higher backtracking activity cases, although a plain parser hardly ends within an acceptable time, our elastic packrat parser achieves even better performance than conventional packrat parsers. These results indicate that the elastic packrat parsing can be the best single choice when implementing PEG-based parsers.

The remainder of the paper is structured as follows. Section 2 states several performance considerations with our preliminary empirical analysis. Section 3 presents the idea of packrat parsing with elastic sliding window by extending a tabular recursive descent parsing. Section 4 demonstrates the experimental results. Section 5 reviews related work. Section 6 concludes the paper.

2. Preliminary Investigation

Many researchers and developers have made packrat parsers with a variety of programming languages. As practitioners' experiences increase, they increasingly question whether the cost of memorization is justified by its benefits. Supplementally, there are many claims in the literature [1], [9], [11], such that plain recursive descent parsers are sufficiently competitive, or even faster than packrat parsers.

More recently, the acceptance of packrat parsing has become more contraversal to many practitioners. We start by investigating several factors that impact the performance of packrat parsing.

2.1 Backtrack Activity

Backtrack activity is a major performance factor of top-down recursive descent parsing [2], including packrat parsing. To begin, we investigate backtracking activity in cases of parsing various formats of input sources. **Figure 1** sketches the backtracking activity in a recursive descent parser; if the matching succeeds, the parser consumes the character stream and then moves its parsing position forward, while it *backtracks*, or moves the position backward to alternatives, if the matching fails. The head position is defined as the maximum consumed length, and the backtracking length is measured from the head position at each backtracking time, since the occurrences of backtracking are usually nested.

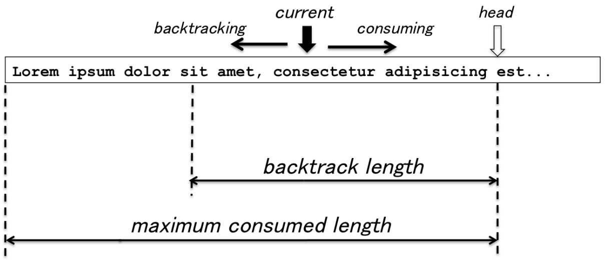


Fig. 1 Backtrack activity.

To assess the backtracking activity, we define the following two indicators:

- *Backtracking ratio* – the ratio of the total length of backtracking to the input size. In other words, we say the length-weighted mean of backtracking per one byte. For example, the ratio 0.0 means no backtracking, and 1.0 means that the parser backtracks as long as it consumes the characters in total.
- *Longest backtrack* – the maximum length of backtracking from the head position, observed through the whole parsing time.

Table 1 shows the summary of our preliminary investigation on backtracking activity. Input sources that we have investigated include C, Java, JavaScript, CSV, XML, JSON, and log formats. The results suggest that backtracking activity depends totally on the format type of the inputs. In parsing CSV files, for instance, no backtracking occurs. On the other hand, a higher backtracking ratio is observed when parsing C and JavaScript sources. In general, we can say that data syntax implies relatively lower activity compared to programming language syntax. This reason is that data formats likely include distinct key characters, such as < and [, by which its succeeding sub-elements can be recognized without backtracking.

Based on our parsing experience, we introduce three levels to describe backtracking activity. We say that the backtracking activity is *low* if the backtracking ratio is less than 1.0. The *moderate* activity is between 1.0 and 10.0, and *high* is more than 10.0. As we will demonstrate in Section 4.3, high backtracking activity corresponds to so-called exponential behaviors, in which packrat parsing is required.

The longest backtrack suggests how long we need to keep memoized results from the head position at each backtrack time. Interestingly, most of the data sets showed very short length compared to their input sizes. As observed in the XMark data sets [10], ranging from 10 MB up to 1 GB, the longest backtrack is constant and not linearly scaled to the size of the inputs; a large scale of data is simply comprised of repeated portion of smaller sub-elements, as defined in `Xml*` and `Statement*`. In such a structure, backtracking rarely occurs across each of the

Table 1 Backtrack activity observed in various inputs.

Source	Format	Rules	Size	Ratio	Worst
NSSCache	C	250	260K	1.2461	227
PEG	C	250	64K	1.4333	6,482
JQuery	JS	326	240K	13.623	247,067
AssureNote	JS	326	1.2M	39.554	70,289
DBLP	XML	24	1.4G	0.159	1
XMark	XML	24	10M	0.085	1
XMark	XML	24	100M	0.085	1
XMark	XML	24	1G	0.089	1
Earthquake	Atom	24	6.0M	0.398	3
Earthquake	JSON	29	6.0M	0.0013	1
Earthquake	CSV	3	1.3M	0	0
JPZIP	CSV	3	8.8M	0	0
Syslog	Log	15	1.3M	0.0412	4

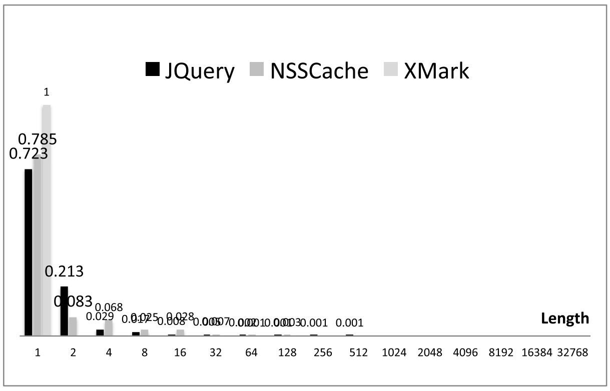


Fig. 2 Histogram of backtracking length: JQuery (formatted in JavaScript), NSSCache (in C), and XMark (in XML).

Table 2 Memoization statistics.

Source	Format	Rules	Memo	Used	Ratio
NSSCache	C	250	93,169	9,094	0.097
PEG	C	250	932,784	156,989	0.168
JQuery	JS	326	1,148,621	193,099	0.168
AssureNote	JS	326	424,594	63,854	0.150
DBLP	XML	24	6,599,423	34,062	0.051
XMark	XML	24	2,895,196	166,631	0.057
Earthquake	Atom	24	1,718,737	89,444	0.052
Earthquake	JSON	29	4,161,232	34,080	0.008
Earthquake	CSV	3	0	0	0
JPZIP	CSV	3	0	0	0
Syslog	Log	15	94,727	8,819	0.093

sub-elements.

Table 1 also suggests that backtracking activity is *localized* in most cases. However, a significant exception is observed in JavaScript cases, such as JQuery and AssureNote. The reason for this is that in order to recognize the semicolon auto-insertion syntax of JavaScript we need to attempt two different semicolon patterns for expressions, resulting in inevitable backtracking. In addition, JavaScript's functional programming style allows blocks and statements to be included in expressions; as a result, `Expression` can dominate most portions of the whole input. Therefore, JQuery's worst activity length almost reaches the whole size of its input.

To investigate the details of backtracking activity, we focus on three typical cases of backtracking activity, including JQuery (high), NSSCache (moderate) and XMark (low). **Figure 2** illustrates the distribution of backtracking lengths. In all cases, 99.9% of backtracking occurs within a range of 16-byte length from the head position, even although the longest backtrack is longer than the 16-byte length.

2.2 Effectiveness of Memoization

The claim "No Memoization Necessary" comes from the fact that most memoized results are surprisingly unused. **Table 2** illustrates this trend by showing the utilization of memoization, where the utilization is simply measured by the ratio of used results to stored ones. Note that these numbers include multiple counts of the same used results, which can also be regarded as the caching hit ratio. In even high backtracking activity cases, such as in JQuery, at most 17% of memoized results are used in total. This suggests that most of the table lookup attempts will fail, resulting in overhead cost that yields no benefit.

Table 3 Top 10 of 326 nonterminals on memoization utilization.

Nonterminal	U/S	Occurrence	Mean	Max
`Spacing`	3.25	0.7239065	0.63	330
`Expression`	1.69	0.1646081	116.63	247,066
`DecimalLiteral`	0.90	0.0535756	0	0
`Catch`	0.85	0.0001051	103.58	386
`Word`	0.64	0.0257852	0	0
`ElisionList`	0.30	0.0013098	0	0
`MemExpression`	0.27	0.0778731	0	0
`Identifier`	0.09	0.0754150	0	0
`Digit`	0.03	0.0542872	0.11	1
`UCHAR`	0.02	0.0877174	0	1

Choosing nonterminals to be memoized is a common strategy in packrat parsing, since the utilization trend varies from nonterminal to nonterminal. To detail this, we focus on the parsing statistics of JQuery and then extract the top 10 utilized nonterminals from 326 nonterminals. **Table 3** summarizes the top 10 nonterminals. The column labeled "U/S" stands for the ratio of used by the stored results. The column labeled "occurrence" indicates how many nonterminals are called per each byte of input. A lower occurrence rate means that memorized results are rarely called in total. The two columns labeled "Mean" and "Max" indicate how long the memoized results avoid the repeated parsing process. While the length 0 may seem strange, we count the length as 0 when the nonterminal calls fail. In summary, a very small number of nonterminals, such as `Spacing` and `Expression`, dominate the effective part of the memoization process in the case of JQuery.

A question arises about how we choose such effective nonterminals in advance. In Refs. [1], [5], there is a little confusion on the choice of nonterminals. For example, the `Spacing` rule is regarded as transient, which means no need for memoization in Ref. [5]. However, as Table 3 indicates, the `Spacing` rule shows the second best utilization. The heuristic analysis based on the properties of a defined grammar is not so easy, and prone to error [1].

Furthermore, another difficulty comes with the efficient implementation of the memoization table. In existing packrat parsers, the memoization table is typically implemented with a two-dimensional array, an array of linked lists, or a hash table, which delivers different performance. Indeed, the effectiveness of the `Spacing` nonterminal relies on whether the table lookup is faster than the repeated attempt of lexical matching. The effective packrat parser requires considerable sophistication in implementing its memoization table.

3. Elastic Packrat Parsing

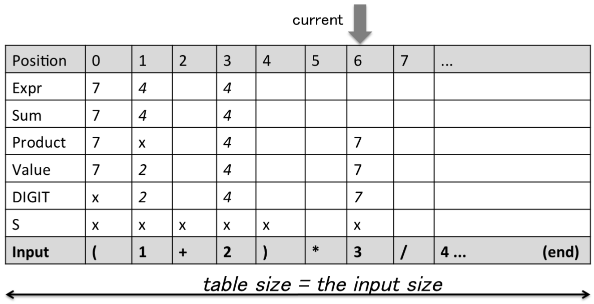
The purpose of this paper is to extend a conventional packrat parsing in a way that achieves stable and robust performance in various parsing contexts, including very large inputs and various levels of backtracking activity. This section describes its extension with three stepped approaches: sliding window, tracing nonterminals, and elastic table structure.

3.1 Packrat Parsing and Memoization Tables

To begin, we revisit how packrat parsing works. As Ford pointed out [3], packrat parsing is a lazy evaluation version of tabular top-down parsing. The tabular parsing uses a memoization

```
Expr    = Sum
Sum     = Product S* (('+' / '-') S* Product)*
Product = Value S* (('*' / '/') S* Value)*
Value   = DIGIT+ S* / '(' S* Expr S* ')' S*
DIGIT   = [0-9]
S       = [ \t]+
```

Fig. 3 Example of a grammar definition in PEGs.



Position	0	1	2	3	4	5	6	7	...
Expr	7	4		4					
Sum	7	4		4					
Product	7	x		4			7		
Value	7	2		4			7		
DIGIT	x	2		4			7		
S	x	x	x	x	x		x		
Input	**(**	**1**	**+**	**2**	**)**	*****	**3**	**/**	**4 ... (end)**

Fig. 4 Memoization table in packrat parsing.

table to avoid redundant nonterminal calls by storing all results of each nonterminal call at each distinct position in the input character. The parsed results are stored in the table, which has one row for each nonterminal and one column for each position. Packrat parsing fills out this table incrementally. **Figure 3** shows a sample grammar definition written in PEGs, and **Fig. 4** sketches a memoization table for calling the nonterminal Sum at the 6th position of the input. The numbers filling cells stand for consuming input characters.

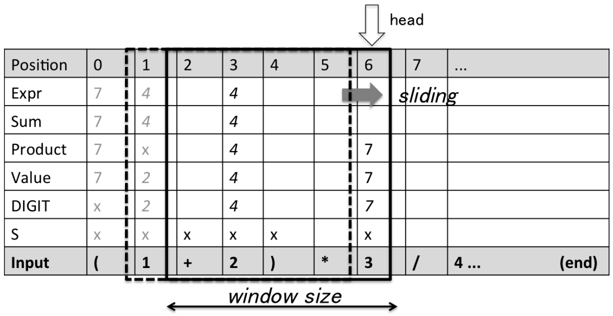
Many packrat parsers have adopted a memory-effective structure, such as an array of linked lists or a hash table of linked lists, instead of a two-dimensional array. Despites this struggle, the space complexity for memoization is $O(L \times N)$, where L is the length of the input and N is the number of nonterminals defined in a given grammar. Decreasing the size of the memoization table is a central issue for improving parser performance.

3.2 Sliding Window

The idea of the sliding window comes from the field of networking protocols and querying streaming data, for which we need to be able to handle an unlimited length of incoming data, such as data streams. A fixed size buffer is allocated as a window, and we slide the window buffer in such a way that it can include newly arrived data while old data outside of the window is expelled from the buffer.

We use the sliding window on a memoization table to reduce memory consumption. The window slides when the head position of the parser moves forward. **Figure 5** sketches a 5-length sliding window for a memoization table. The rightmost position of the sliding window is always equal to the head position of the parser.

An arising problem is how to set an appropriate window size. Apparently, the longest backtrack, described in Section 2.1, is the best candidate for that purpose, because all necessary memoized results can be kept to avoid redundant calls. Unfortunately, it is impossible to estimate the longest backtrack from a defined set of grammar rules. This is because the length that will be consumed in each nonterminal call depends largely on the contents of the input. In theory, the longest backtrack can be the input size itself, which means the same as the conventional non-sliding memoization table.



Position	0	1	2	3	4	5	6	7	...
Expr	7	4		4					
Sum	7	4		4					
Product	7	x		4			7		
Value	7	2		4			7		
DIGIT	x	2		4			7		
S	x	x	x	x	x		x		
Input	**(**	**1**	**+**	**2**	**)**	*****	**3**	**/**	**4 ... (end)**

Fig. 5 Sliding window for memoization table.

As we investigated in Section 2.1, most occurrences of backtracking are intensely localized and appear within a very closed range from the head position. For example, the 16-byte window size allows to avoid redundant calls when 99.9% of backtracking occurs. We consider that the shortened window size may yield measurable benefits.

The clear advantage is that the sliding window can guarantee the upper limit of heap consumption for memoization. The guarantee of linear time parsing is also preserved if the preset window size is smaller than the longest backtrack. An obvious problem is that rapid degradations are expected when the backtracking length exceeds the preset size of sliding window.

3.3 Tracing and Dynamic Deactivation

Choosing nonterminals to be memoized is another common approach to improving packrat parsing. However, as we described in Section 2.2, the static analysis is not so easy and is prone to errors. That is why we attempt the dynamic analysis by tracing memoization behaviors.

Our idea is straightforward. The parser starts off with activated memoization for all nonterminal calls. We trace some performance factors of each nonterminal call, and then deactivate its memoization if the traced factors fall below the expected performance. In this paper, we simply trace the utilization ratio of nonterminal memoization, described in Section 2.3, although we have attempted several cost-based metrics. Since an earlier judgment of the deactivation yields bigger benefits, we check whether the first 32 calls include at least one used result. If not, the memoization is deactivated.

The clear strength of our dynamic analysis is that it is simple and easier to implement than any static analysis methods. It also helps avoid heuristic errors. On the other hand, the drawback is that it requires some dynamic data structure to store a variable number of nonterminals. This may lead to a slower table lookup compared to the static removal of ineffective nonterminals.

3.4 Elastic Packrat Parsing

Elastic packrat parsing is a hybrid version of the sliding window and the dynamic analysis approach, which are orthogonal to each other. While the combination seems trivial, we have made

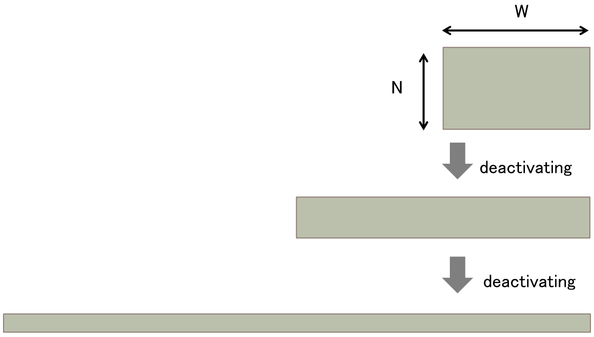


Fig. 6 Concept of elastic sliding window.

several refinements for a synergetic purpose. **Figure 6** illustrates the concept of elastic sliding window, where the window size is expanded as the nonterminals are deactivated.

To implement the elastic window, we use a flat-structured array in $W \times N$ length, where W is the window size and N is the number of nonterminals. In principle, the structure of the memoization table maps from a (position, nonterminal) pair to the result of the associated nonterminal call at the associated position. Accordingly, the flattened array potentially has the same space capacity with the $W \times N$ table.

Next, we use a hashing-based index to locate where the memoized result can be found on the array. The unique key is generated from the position and the nonterminal. We reduce the key to an array index by modulo ($W \times N$). The advantage is that the deactivated space could be used by other nonterminals as if the window size were expanded. That is, in an extreme case where only one activated rule remains, the window size is virtually equal to the $W \times N$ length. The drawback is that a collision occurs in two entries having the same index and different keys. This decreases the rigid storage guarantee for the given window size. However, as we demonstrate in the experiments, the length-based guarantee is not so crucial in practice. Rather, we consider that graceful behaviors against the window size are desirable.

Finally, we abandon the explicit sliding expiration to simplify its implementation. The older entry is simply replaced when a newer entry comes. Since new entries are mostly created near the head position, replaced entries are recorded at relatively previous position. This approximately emulates the window sliding operation. The exact emulation costs too much and, furthermore, there is no need to ensure the expiration of older entries in contexts of packrat parsing.

Figure 7 shows the memoize/lookup algorithm of elastic packrat parsing. This demonstrates that a very simple modification of conventional packrat parsing enables our elastic algorithm. Note that the array size in practice is around to a nearby prime number for better hashing results.

4. Evaluation

4.1 Parser Implementation

Nez parser is a parsing runtime based on the Nez grammar language [6], an extended version of PEGs with trans-parsing capability that allows the flexible construction of ASTs. All grammars used in this experiment are written in Nez. However, we turned off the AST construction because we focus more on the parsing performance. The reader may read as if the grammars were described in plain PEGs.

```
//Array: 1-dimensional array for memoized results
//Shift: derived from the binary logarithm of N
def Memoize(Position, NonTerminal, Result):
  key = (Position << Shift | NonTerminal )
  index = key % (W * N)
  array[index].key = key
  array[index].result = Result

def Lookup (Position, NonTerminal, Result):
  key = (Position << Shift | NonTerminal )
  index = key % (W * N)
  return (array[index].key == key)
                ? array[index] : null
```

Fig. 7 Memoization algorithm of elastic packrat parsing.

The Nez parser is designed to work as a dynamic parser – say, an interpreter-based parser of PEGs. In general, dynamic parsing is usually slower than a statically generated and optimized parser with C and Java. This is, in part, why we do not perform a comparative study with other PEG-based parser generators in this paper.

The Nez parser is written in Java. The experimental elastic packrat parser and other comparative parsers are implemented as the variations of memoization tables. The working source repository of Nez is on the github site, available at `http://nez-peg.github.io/`.

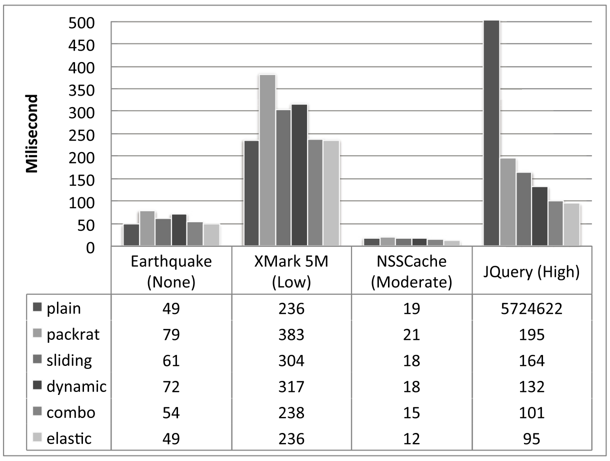
4.2 Experimental Setups

We perform a comparative study with the following set of algorithms. The algorithms are labeled, such as plain and packrat, for readability.

- plain (none) – a plain recursive descent parser without any memoization support.
- packrat (hash, linked-list) – a normal packrat parser that build on a hash table. (Note that we use a hash table because large data sets cause the allocation problem.)
- sliding (array, linked-list) – a packrat parser memoizing on a sliding window in a fixed size. The window size is set to 64 bytes.
- dynamic (hash, linked-list) – a packrat parser tracing memoization behaviors to deactivated nonterminals, as described in Section 3.3.
- combo (array, linked-list) – a packrat parser that combines the sliding window with the dynamic deactivation.
- elastic (array) – an elastic packrat parser implemented as the algorithm described in Section 3.4.

Backtracking is a major factor that has a substantial impact on the performance of packrat parsing [1], [3]. In addition, the longest backtrack is perhaps a potential factor do argue the window size. To investigate these factors, we have chosen data sets from Table 1 to include a variety of backtracking activities. The data sets investigated are also labeled as follows:

- Earthquake (formatted in CSV) – the open data obtained from data.gov. In this data sets, no backtracking occurs when parsing.



	Earthquake (None)	XMark 5M (Low)	NSSCache (Moderate)	JQuery (High)
■ plain	49	236	19	5724622
■ packrat	79	383	21	195
■ sliding	61	304	18	164
■ dynamic	72	317	18	132
■ combo	54	238	15	101
■ elastic	49	236	12	95

Fig. 8 Latency in various backtracking activities.

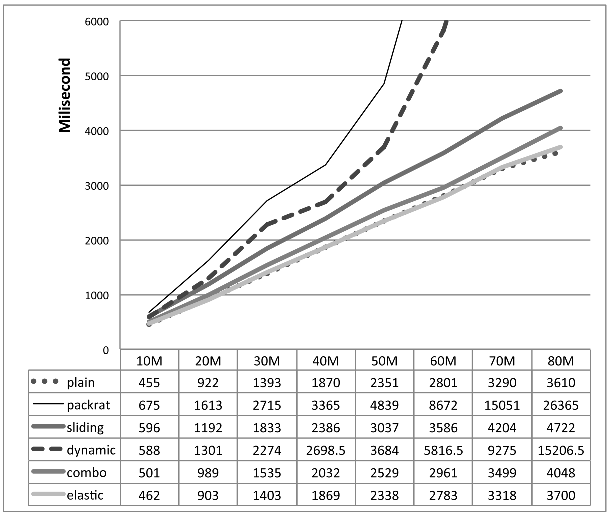
- XMark (formatted in XML) – synthetic and scalable XML files that are provided by XMark benchmark program [10]. The backtrack activity is at the low level (ratio 0.085).
- NSSCache (formatted in C) – a preprocessed C source code that comes from Google's NSS Cache project. The backtracking activity is at the moderate level (ratio 1.24).
- JQuery (formatted in JavaScript) – a popular JavaScript library. To analyze some larger backtracking length, we use an uncompressed version. The backtracking activity is at the high level (ratio 13.62) and the longest backtrack exceeds the practical size of the sliding window.

The test environment is an Apple MacBook Air, with 2 GHz Intel Core i5, 4 MB of L3 Cache, 8 GB of DDR3 RAM, running Mac OS X 10.8.5 and Oracle Java Development Kit version 1.8. The initial heap size (-Xms) is set to 2 GB to decrease garbage collector's side effects. All measurements represent the best result of five or more iterations over the same input.

4.3 Performance

Figure 8 shows the effects of a memoization table by comparing the latency measured in milliseconds with six different parsers for each data set. With low backtracking ratios, such as in Earthquake and XMark, the plain parser is clearly the fastest because almost all memoization efforts are unnecessary. However, the combo and elastic parsers show a very competitive performance to the plain parser. Note that most of the nonterminals are deactivated at the earlier position of parsing. Unfortunately, the dynamic parser does not achieve significant speedups, perhaps due to the different implantation of its table lookups.

We will turn to the test result NSSCache in the moderate backtracking activity case. The plain parser is still faster than the packrat parser, while the other four enhanced packrat parsers achieve some improved performance. This suggests that the memoization support is effective, but not mandatory at this level of backtracking activity. We confirm that this is the same situation in which developers have faced the traditional algorithm choice. In the high backtracking activity case shown in JQuery, the plain parser is several orders of magnitude slower than the packrat parser. More than 10-minute latency would be unacceptable in any practical parsers.



	10M	20M	30M	40M	50M	60M	70M	80M
• • • plain	455	922	1393	1870	2351	2801	3290	3610
packrat	675	1613	2715	3365	4839	8672	15051	26365
sliding	596	1192	1833	2386	3037	3586	4204	4722
dynamic	588	1301	2274	2698.5	3684	5816.5	9275	15206.5
combo	501	989	1535	2032	2529	2961	3499	4048
elastic	462	903	1403	1869	2338	2783	3318	3700

Fig. 9 Latency in scalable XMark files (10–80M).

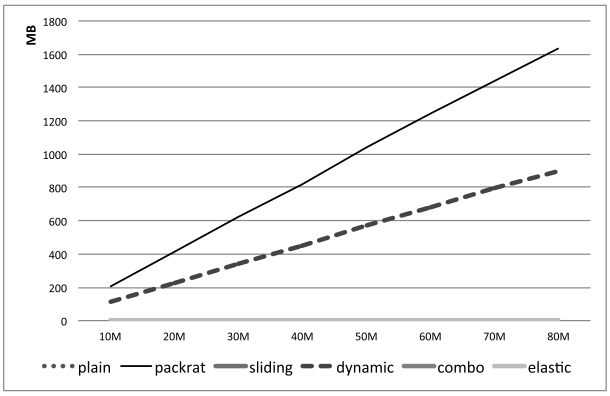


Fig. 10 Heap consumption in scalable XMark files (10–80M).

The elastic packrat parser demonstrates the best performance at all backtracking levels. Compared to the combo parser, our refinements on the elastic table create measurable improvements.

Next, we will focus on memory pressure when parsing the megabyte order of inputs. **Figure 9** shows the latency in various sizes of XMark-generated files, scaling from 10MB up to 80 MB. **Figure 10** shows the corresponding heap consumption. In theory, the normal packrat parser requires linear heap consumption in relation to the input size and Fig. 10 confirms such heap consumption in reality. More importantly, we found the super-linear time behavior in over 30 MB files. This indicates that the memory pressure invalidates the packrat parsing's linear time performance. The dynamic parser decreases the effects of memory pressure, but does not solve the problem. The three window-based packrat parsers successfully keep the constant heap consumption and show no significant performance degradation.

We have concluded that the elastic packrat parsing maintains robust and competitive performance in various parsing contexts, including high backtracking activity and large-scale input sources.

4.4 Memoization Effects with Various Window Sizes

As we have predicated in Section 3, a small window size can cause invalidated memorization. We investigate the impact of the window size by scaling it up to a 1,024-byte length, where the

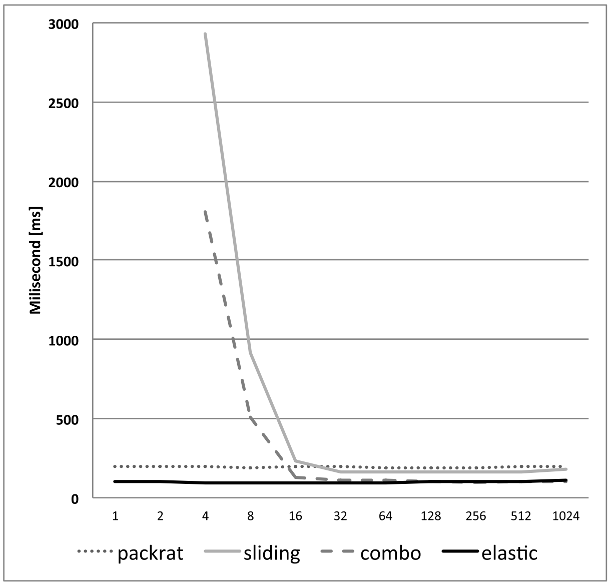


Fig. 11 Latency in various window sizes.

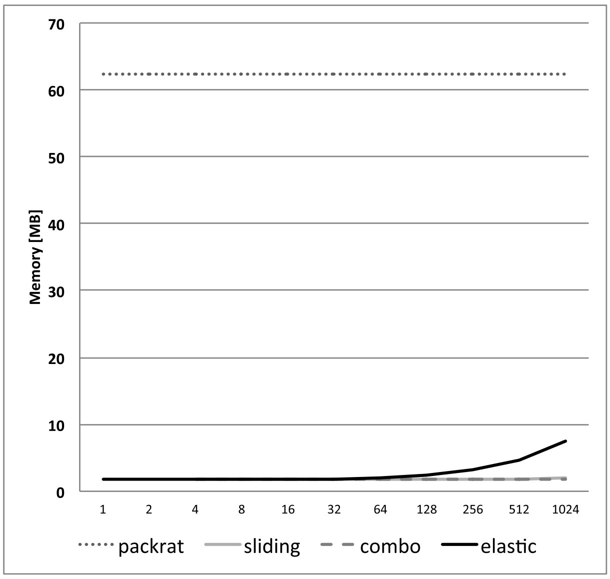


Fig. 12 Heap consumption in various window sizes.

effect seems stable. The data set that we investigated is JQuery, simply because its longest backtrack (247,067 bytes) exceeds the window size to store all memoized results. **Figure 11** shows the latency in each experimental window size and **Fig. 12** illustrates the corresponding heap consumption. In this examination, we compared three window-based parsers (labeled sliding, combo, and elastic) with packrat parsers. Note that the packrat parser causes no invalidated memoization in either theory or practice.

As we expected, both sliding and combo parsers show significant performance degradations, especially in under 16-byte window sizes. Note that we omit data points for 1 and 2 that take too much time to parse. The performance, however, is rapidly improved at the data point of window sizes over 32-bytes, which is even smaller than the longest backtrack.

The elastic parser, on the other hand, can achieve a stable performance even at 1-byte window size. In fact, the larger window sizes result in minor performance degradation (from the best, 91 ms in 8-byte length, to the worst, 113 ms in 1,024-byte length), probably due to the L2 cache efficiency. Soaring memory consumption have found only in elastic packrat parsing. This is because that sliding and combo use a linked list to store nonterminal results, allowing a sparse table. In either case, the total heap consumption can remain at considerably lower levels compared to the packrat parser.

This experiment result leads to the conclusion that the longest backtrack is not required to set the window size. Perhaps, a window size that can cover 99% of backtracks is long enough in practice. As long as we have performed the empirical study (as shown in Section 2), backtracking activities are usually localized, which makes it easier to choose a practical window size. Indeed, our Nez parser is set to a 64-byte length by default.

4.5 On Complexity

The strength of packrat parsing is its linear parsing time guarantee, based on the ground that all intermediate results are memoized to avoid redundant calls. In turn, the elastic packrat parsing may expire memorized results that are stored outside the sliding window. However, it still ensures a linear time parsing if the longest backtrack is smaller than the window size. Since backtracking out of the window rarely happens, we say that elastic packrat parsing is a mostly linear time parsing algorithm.

The performance impact of “out of the window” backtracking is still unfamiliar. Intuitively, backtracking that reaches nearly the input size would double the total parsing length, and probably leads to a significantly increased parsing time compared to conventional packrat parsing. However, no such an increase is observed in the JQuery case, although several times of “out of the window” backtracking are included. We have no idea for the exact reason, but the impact of “out of the window” backtracking is immeasurable in practice.

Expanding an window size at runtime can be a reasonable attempt for better time guarantee. However, we abandoned this attempt, because of the possible loss of the constant space guarantee. As the JQuery cases indicates, the longest backtrack may reach the input size itself. The constant space guarantee is another desirable property in parsing. In conclusion, the elastic packrat parsing archives the best balance between time and space complexity.

5. Related Work

Packrat parsing is a popular technique for implementing recursive descent parsers with backtracking. Two seminal packrat parsers, Ford’s Pappy [3] and Grimm’s Rats! [5], have been successfully developed and shown the technique’s linear parsing time. Unfortunately, they did not perform comparative studies with a plain recursive descent parser, perhaps due to the fact that the superiority of packrat parsing seemed trivial in their experiments. However, as practical experiences have increased in various parsing contexts, it is no longer so trivial.

In Refs. [1], [9], [11], many researchers doubted the effectiveness of memoization in practice. Warth et al., the developers of OMeta [11], pointed out that the overhead of memoization may outweigh its benefits for the common case. In addition, Becket et al., based on their definite clause grammars (DCGs) analysis, concluded that packrat parsing might actually be significantly less efficient than plain recursive descent parsers with backtracking [1]. Medeiros et al. proposed an alternative method to packrat parsing [7]. Their claims are based on the assumption that exponential behavior does not happen in practice. However, as we have shown before, the parsing of a JQuery library rebuts this assumption.

The huge heap consumption in memoization has been recognized as the crucial problem with packrat parsing. Mizushima et al. extended some PEG notion with a cut operator in order to directly limit the backtracking region, and presented mostly constant heap consumption with their parser generator, Yapp [8]. Interestingly, as the paper presented the automatic insertion of cut operations, packrat parsing can perhaps adopt their idea as a part of its optimization process. However, they have not argued for the applicability of cut operations in high backtracking activity cases.

Redziejowski, the author of Mouse, focused on the number of nonterminal calls instead of the input size, and concluded that memoizing the two most recent results succeeded in significantly reducing heap consumption [9]. This approach is similar in terms of expiring older memoized results, while Mouse's approach abandons the linear time parsing property. In our approach, we demonstrate that elastic packrat parsing can handle potential exponential cases.

Choosing nonterminals to be memoized was first attempted as a part of Rats! optimization [5]. The grammar developers can specify the transient attribute to apply their heuristics to control whether nonterminals will be memoized or not. Becket et al. pointed out the heuristic analysis of grammatical properties resulted in very limited improvement [1]. Our dynamic analysis approach, on the contrary, avoids such difficulties and yields clear effects for improved performance.

6. Conclusion

Packrat parsing is a popular technique for implementing recursive descent parsers with backtracking. However, many practitioners have found the total performance of packrat parsing is questionable in comparison to plain recursive descent parsers. Some of them have even abandoned memoization and its potential benefit: the linear time parsing guarantee. This hard choice is mainly due to the unstable performance of packrat parsing in relation to the size of the input source and its underlying backtrack activity.

This paper showed that the simple modification of packrat parsing makes it more stable in terms of input variety and achieves competitive performance compared to both conventional packrat parsers in exponential cases and plain parsers in very low backtrack activity cases. Accordingly, we provide the best single choice for practitioners who attempt PEG-based parsing.

Acknowledgments Tesuro Matsumura implemented the JavaScript grammar and Atsushi Uchida and Maskaki Ishii implemented the C grammar. The authors thank the IPSJ/SIGPRO members for their feedback and discussions.

References

[1] Becket, R. and Somogyi, Z.: DCGs + Memoing = Packrat Parsing but is It Worth It?, *Proc. 10th International Conference on Practical Aspects of Declarative Languages*, *PADL '08*, pp.182–196, Springer-Verlag, Berlin, Heidelberg (online), available from ⟨http://dl.acm.org/citation.cfm?id=1785754.1785767⟩ (2008).

[2] Birman, A. and Ullman, J.D.: Parsing algorithms with backtrack, *Information and Control*, Vol.23, No.1, pp.1–34 (online), DOI: http://dx.doi.org/10.1016/S0019-9958(73)90851-6 (1973).

[3] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, *ICFP '02*, pp.36–47, ACM (online), DOI: 10.1145/581478.581483 (2002).

[4] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, *POPL '04*, pp.111–122, ACM (online), DOI: 10.1145/964001.964011 (2004).

[5] Grimm, R.: Better Extensibility Through Modular Syntax, *Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '06*, pp.38–51, ACM (online), DOI: 10.1145/1133981.1133987 (2006).

[6] Kuramitsu, K.: Toward Trans-Parsing with Nez, *Informal Proceedings of JSSST Workshop on Programming and Programming Languages* (2015).

[7] Medeiros, S. and Ierusalimschy, R.: A Parsing Machine for PEGs, *Proc. 2008 Symposium on Dynamic Languages*, *DLS '08*, pp.2:1–2:12, ACM (online), DOI: 10.1145/1408681.1408683 (2008).

[8] Mizushima, K., Maeda, A. and Yamaguchi, Y.: Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space, *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, *PASTE '10*, pp.29–36, ACM (online), DOI: 10.1145/1806672.1806679 (2010).

[9] Redziejowski, R.R.: Parsing Expression Grammar As a Primitive Recursive-Descent Parser with Backtracking, *Fundam. Inf.*, Vol.79, No.3-4, pp.513–524 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2367396.2367415⟩ (2007).

[10] Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I. and Busse, R.: XMark: A Benchmark for XML Data Management, *Proc. 28th International Conference on Very Large Data Bases*, *VLDB '02*, pp.974–985, VLDB Endowment (online), available from ⟨http://dl.acm.org/citation.cfm?id=1287369.1287455⟩ (2002).

[11] Warth, A. and Piumarta, I.: OMeta: An Object-oriented Language for Pattern Matching, *Proc. 2007 Symposium on Dynamic Languages*, *DLS '07*, pp.11–19, ACM (online), DOI: 10.1145/1297081.1297086 (2007).

Kimio Kuramitsu is an Associate Professor, leading the Software Assurance research group at Yokohama National University. His research interests range from programming language design, software engineering to data engineering, ubiquitous and dependable computing. He has received the Yamashita Memorial Research Award at IPSJ. His pedagogical achievements include Konoha and Aspen, the first programming exercise environment for novice students. He earned his B.E. at the University of Tokyo, and his Ph.D. at the University of Tokyo under the supervision of Prof. Ken Sakamura.