International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Optimization Strategies for Performance Enhancement of Packrat Parsers

Nikhil Mangrulkar*[1], Kavita Singh[2], Sagar Badhiye[3]

Submitted: 22/12/2023 **Revised**: 28/01/2024 **Accepted**: 08/02/2024

Abstract: Memoization in computing refers to storing intermediate results and referring them when same inputs appear again instead of calculating them again. Packrat parsing is a comparatively new parsing technique based on top down approach with backtracking for parsing input which uses memoization and ensures linear time parsing. Introduced in 2002 by Bryan Ford, Packrat parsing was developed with focus on computer oriented languages. The ensured linear time parsing by packrat parsers comes at a cost of huge primary memory consumption for memoization making it impractical to implement. Here we have proposed a three way approach to optimize the use of memory required for memoization. Our implementation allocates memory for memoization dynamically based on resources available. Non linear data structure eliminates the requirement of continues blocks of free memory. Using linear time searching technique it is ensured that latency is constant even in case where higher number of intermediate results are stored. The proposed implementation is a promising approach for exploiting benefits of memoization to ensure linear time parsing while avoiding burdening the system in case where primary memory is a constraint..

Keywords: *Parsing, Parsing expression grammar, Packrat parser, Linear time parsing, Memoization*

1. Introduction

Parsing in computer science is a part of language processing and is used to decide whether the input string is according to the syntax of that programming language or not. Typically, parsing comprises of two tasks: lexical analysis and parsing. Lexical analysis is the process of dividing the input string into smallest recognizable units called lexemes and create tokens of those lexemes which are in form of <token-name, attribute-value>. These tokens are provided to the parser for further processing. The parser refers to the rules of the language which are written using grammar, to decide whether input string can be accepted by the language.

The initial research on parsing was focused on parsing natural or human language [1]. The ability of regular expression (RE) and context free grammar (CFG) to represent ambiguity was the obvious reason for their use to specify the rules for parsing natural language. It was observed later that the method used for parsing natural language could also be used for parsing machine oriented or computer programming languages. The principal difference between computer programming language and natural language is that most of the programming languages are designed not to be ambiguous. Thus, the ability of RE's and CFG's to represent ambiguity is not required in computer-oriented languages. Instead, sometimes it becomes overhead to handle the ambiguity where it was not required in first place. It is very well-known that top-down parsing methods with backtracking struggle with two major issues: First, a top-down parser generally fails to terminate on some inputs while using a left-recursive grammar. Second, in backtracking parsers a noticeable amount of redundant computation is required, and in the worst case, parsing time is exponential in the length of the input string [2].

Several different parsing techniques based on various approaches like top-down parsing and bottom-up parsing have been developed by researchers. Packrat parsing is one of such relatively newer technique developed by Bryan Ford keeping focus on machine-oriented languages[3] [4]. Packrat parsing is a backtracking supportive top-down approach for parsing inputs. It uses Parsing Expression Grammar (PEG) instead of RE's or CFG's to specify the rules. PEG was also introduced by Bryan Ford during his work on packrat parsers [1]. The initial work was done by A. Birman et. al. [5] which was further worked upon by Aho and Ullman and called is generalized top-down parsing language (GTDPL). This was the first deterministic backtracking top-down parsing algorithm. Due to the deterministic nature of resulting grammar, it was found that the parsing results could be memoized to avoid redundant calculations. Memoization in computer science is the technique used to store all intermediate results and refer them whenever same calculations come up in future. But at that time, availability of main memory was limited, because of which this approach was never practically implemented.

The main issue with packrat parsing is the requirement of huge memory for memoization. As correctly pointed out by

[1] *Yeshwantrao Chavan College of Engineering, Nagpur – 441110, INDIA*
ORCID ID : 0000-0001-5190-9722
[2] *Yeshwantrao Chavan College of Engineering, Nagpur – 441110, INDIA*
ORCID ID : 0000-0003-4012-6786
[3] *Symbiosis Institute of Tech. Nagpur Campus, Nagpur– 440008, INDIA*
ORCID ID : 0000-0002-0710-3761
* *Corresponding Author Email: nmangrulkar@ycce.edu*

Ralph Becket and Zoltan Somogyi, for every byte of the input, about 400 bytes of memory is required [6]. The memory required for memoization is directly proportional to the input and as the size of input grows, memory requirement of memoization increases linearly, raising the question on benefits provided by memoization over the resources required to make it possible.

In this paper, rather than avoiding memoization completely, we propose a three-way approach for optimizing the use of memory required for memoization, making it possible to exploit the benefits that are offered by memoization when possible, while avoiding the excessive use of memory when resources are limited.

2. Related Work

Conventional top-down parsers with backtracking may face exponential parsing time in case of backtracking. Because of the fact that packrat parsers can achieve linear time parsing even in the case of backtracking, researchers have put the efforts into making packrat parsers implementable. For a detailed literature review on packrat parsers, our review paper [7] on this topic can be referred. Here, only the work carried out to improve the performance of packrat parsers has been discussed.

Robert Grimm presented the packrat parser-based Rats! parser generator for Java that translates a grammar specification into programming language source code. [8] Rats! ensures linear-time performance as it stores intermediate results. The paper emphasizes the importance of parsers as a essential first step for any language processor and compares the difficulties in extending context-free grammars and LR or LL parsers with the advantages of using packrat parsers. The experimental evaluation results demonstrating the parser generator's performance and usability has also been described in the paper.

Alessandro Warth et. al. in their presented work discussed about the modification of packrat parsers to support direct and indirect left recursion without the need for left recursion elimination transformations [9]. The given modifications enable packrat parsers to parse a broader class of grammar and extend their support to left-recursive portions of grammar, such as Java grammar. The authors also discuss the impact of these modifications on parse times.

Mouse, a tool for transcribing Parsing Expression Grammar into an executable parser in Java is presented by Roman R. Redziejowski [10]. The paper explains how to define parsing expressions in PEG, how Mouse generates a parser from the specified grammar, and how it incorporates semantic actions to provide meaning to parsed structures. It also discusses the efficiency trade-offs involved in using backtracking, the possibility of memoization to improve performance.

Manish M. Goswami, et. al., presented technique for improving the performance of a stack-based recursive-descent parser for Parsing Expression Grammar [11]. The authors have proposed optimizing a stack-based recursive-descent parser by eliminating function calls for grammar production and using stack operations instead. Moreover, they introduce optimizations using the * (star) and cut operators, which help reduce redundant stack operations and backtracking activities, respectively. The experimental results presented shows that the optimized stack-based parser offers better performance over a straightforward recursive-descent parser and competitive performance when compared to packrat parsers with memoization. The comparison includes metrics like the number of packrat pushes, backtrack pushes, CPU time, and backtracking activity.

Kimio Kuramitsu tried to address the issue of high memory consumption associated with packrat parsing by introducing the concept called elastic packrat parsing, which employs a sliding window buffer to store memoized results, thereby bounding heap consumption and maintaining constant space complexity regardless of input size [12]. The presented approach leverages the observation of worst-case backtrack lengths to determine the buffer's size. Since it's challenging to know the longest backtrack length before parsing, an approximated window size is selected based on empirical analysis, which can then be adjusted during parsing.

3. Proposed Methodology

Here we are presenting a three step approach for optimizing memory usage for storing intermediate results.

3.1. Dynamic Memory Allocation for Memoization

Even though memoization offers huge advantage of guaranteed linear time parsing over traditional top town parsers with no memoization, packrat parsing is still not adapted at large. The main reason behind this is the huge memory consumption for memoization as mentioned earlier. For this reason alone, many researchers have criticized memoization due to the cost at which it comes. With the advancement of technology, memory may be available in sufficient size in some machines.

Here we propose an algorithm to use memory optimally to take most of the advantage that memoization has to offer. We have implemented logic to cap the limit on memory usage for memoization dynamically. If sufficiently large memory is available in machine, we use memoization without any restrictions. For deciding value of sufficiently large memory (α), we calculate the total free memory available at the time of parsing and if only 1% of it is used to store the intermediate results (1).

$$\alpha = (\mu - \gamma)\, 0.1 \qquad (1)$$

Where,

α is maximum memory that can be used for memoization

μ is total memory available

γ is memory currently being used by the system

The number of results that can be stored in α is the maximum number of intermediate results that will be saved. Maximum memoization is capped to 100 results as it is sufficiently large number of intermediate results for any input. Capping the memory utilization for memoization to α ensures that there is no any additional overhead to the system while ensuring parsing in linear time. For further details of this implementation, our work presented in [13] can be referred.

3.2. Using Nonlinear Data Structure

Parser generator tools like Rats! [8] and Mouse [10] have used array data structure for implementing memoization. Stack based recursive descent parser is also presented [11]. It is commonly known that both array and stack are linear data structures and need continues block of free memory. Situation in which primary memory is available but is scattered, or less memory is available, swapping is required. Swapping in computer science is the technique used to move a process to secondary memory so that more memory becomes available to the other demanding process. It is obvious that swapping may introduce issues of page fault, thereby increasing the process running time.

Using nonlinear data structure is a promising approach specifically in cases where space in main memory is available but is fragmented. Nonlinear data structure does not require continues free memory and we can use fragmented memory instead of implementing sweeping thereby reducing the chance of page faults. We have implemented memoization using Map data structure. Map stores data in combination of <key, value> pair, making it ideal for implementing memoization as entries in memoization are of symbol appeared and number of tokens that can be skipped without rescanning. This information can be efficiently stored in Map data structure.

3.3. Implementing Constant Time Searching Technique

Changing data structure to implement inherently calls for change in searching technique implemented to search stored results. Current implementations where arrays and stack data structure are used to implement memoization, binary search technique is used. The searching time in binary search is directly proportional to the number of results to be searched. In implementation where array data structure is used, the requirement of binary search of sorted array makes the actual searching complexity of O log(n) as O nlog(n). Stack based implementation has searching complexity of O log(n) and shows better performance over array based implementation.

The number of intermediate results stored in array based and stack based Mouse parser is 10 (optional and maximum) and 5 respectively. This number is very small compared to the maximum number of intermediate results that can be stored in our proposed approach. Therefore, we have implemented searching technique of Hashing, which have constant lookup time of O(1) for any number of results to be looked up as opposed to other implemented searching technique where searching time is proportional to number of results stored. In scenario where memoization is less, Hashing might not perform at par with binary search and might even perform poorer than the later. But as size of memoization increases, hashing ultimately outperforms other implementations. For more details of this proposed implementation, our work presented in [14] can be referred.

4. Implementation

We have implemented our proposed approach by changing the current implementation of Mose tool. Execution has been analyzed on a system with Intel(R) Core(TM) i5-8265U CPU @1.60GHz and 8.00 GB (7.89 GB usable) primary memory. Modifications have been implemented in Java version 1.8.0_331 on Windows 11 OS. The performance of the tool with proposed modifications is stable and is performing accurately under varying input lengths.

5. Result & Discussion

To evaluate the performance of our proposed approach as compared to other implementations, various parameters are used for comparison. Heap utilization, time required to search intermediate results and time complexity of searching techniques used by various implementations.

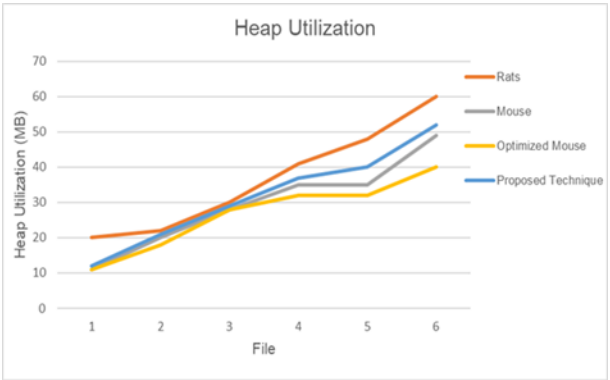

Fig 1. Heap utilization by various implementations.

Fig. 1 shows memory consumption by various implementations. In terms of heap consumption, the proposed approach uses more memory as compared to original Mouse tool and Mouse with stack implementation but less than Rats! tool. The extra memory required is due to additional information of

pointers that needs to be stored for Map data structure.

In terms of latency, i.e. time required to access the store intermediate results, as can be observed in fig. 2, is minimum for our proposed approach.

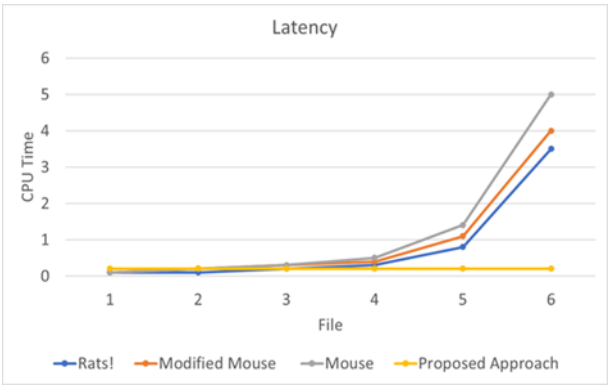

Fig 2. Latency of proposed approach < Rats! < Stack Based Mouse < Mouse.

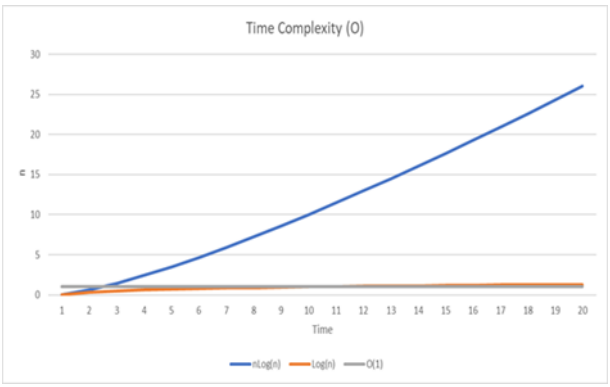

Fig 3. Time complexity of different search techniques.

Figure 3 shows the time complexities of different searching techniques. It can be observed from the figure that when number of intermediate results stored are less, using hashing technique performs poorly than binary search for array and binary search for stack. But as the number of intermediate results to be stored increases, Hashing has a slight upper hand on binary search implemented on stacked based mouse tool.

6. Conclusion and Future Scope

As we understand, the main reason behind non adaption of packrat parser at large is huge memory requirement for implementing memoization. Although memoization guarantees parsing in linear time even in the case of backtracking, the memory resources it consumes is a matter of concern. In this paper we presented a three-way approach for optimizing memory usage required for memoization. Firstly, by using dynamic buffer allocation, we ensure that memoization will not be an additional burden on the system. This also ensures that whenever enough memory is available, maximum memoization will be done and if there is scarcity of primary memory, less intermediate results will be stored. Secondly, by using non linear data structure, Map, fragmented memory can be used for storing intermediate results thereby reducing the page faults and swapping. Lastly, by implementing constant time searching technique, the latency of system is kept constant even in case of large number of intermediate results to be searched for.

The performance of the proposed approach is stable it performs well in terms of latency. Memory required by proposed approach is slightly higher, but it is ensured that it won't prove to be a burden on system.

Further actual reduction in page faults and swapping can be calculated to show the effectiveness of proposed approach.

Author contributions

Nikhil Mangrulkar: Conceptualization, Methodology, Software **Kavita Singh:** Writing-Reviewing and Editing

Sagar Badhiye: Visualization, Validation and Editing.

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," Conference Record of the Annual ACM Symposium on Principles of Programming Languages, vol. 31, pp. 111–122, 2004.

[2] M. Johnson, "Memoization of Top-down Parsing," Computational Linguistics, vol. 21, no. 3, 1995.

[3] B. Ford, "Packrat Parsing: Simple, Powerful, Lazy, Linear Time Functional Pearl," in Seventh ACM SIGPLAN international conference on Functional programming (ICFP '02), New York: Association for Computing Machinery, 2002, pp. 36–47. [Online]. Available: http://pdos.lcs.mit.edu/

[4] B. Ford, "Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking," Dissertation for Master of Science in Computer Science and Engineering, Massachusetts Institute of Technology, 2002.

[5] Birman Alexander, "The TMG Recognition Schema," Dissertation for Doctor of Philosophy, Princeton University, 1970.

[6] R. Becket and Z. Somogyi, "DCGs + Memoing = Packrat Parsing But is it worth it?," in Practical Aspects of Declarative Languages. PADL 2008, P. Hudak and D. S. Warren, Eds., Springer, Berlin, Heidelberg, 2008. doi: https://doi.org/10.1007/978-3-540-77442-6_13.

[7] N. S. Mangrulkar, K. R. Singh, and M. M.

Raghuwanshi, “Parsing Expression Grammar and Packrat Parsing—A Review,” 2023, pp. 377–384. doi: 10.1007/978-981-19-0095-2_36.

[8] R. Grimm, “Practical Packrat Parsing.” [Online]. Available: http://www.cs.nyu.edu/rgrimm/xtc/.

[9] Warth Alessandro, Douglass James R., and Millstein Todd, “Packrat Parsers Can Support Left Recursion,” Association for Computing Machinery, 2010, p. 158.

[10] R. R. Redziejowski, “Mouse: From Parsing Expressions to a Practical Parser,” in CS&P Workshop, Jan. 2009, pp. 514–525. [Online]. Available: http://www.romanredz.se/freesoft.htm.

[11] M. M. Goswami, M. M. Raghuwanshi, and L. Malik, “Performance Improvement of Stack Based Recursive-Descent Parser for Parsing Expression Grammar,” International Journal of Latest Trends in Engineering and Technology, vol. 6, no. 3, pp. 302–309, 2016.

[12] K. Kuramitsu, “Packrat parsing with elastic sliding window,” Journal of Information Processing, vol. 23, no. 4, pp. 505–512, Jul. 2015, doi: 10.2197/ipsjjip.23.505.

[13] Nikhil Mangrulkar, Kavita Singh, and Mukesh Raghuwanshi, “Packrat Parsing with Dynamic Buffer Allocation,” JOURNAL OF ADVANCED APPLIED SCIENTIFIC RESEARCH, vol. 4, no. 1, Apr. 2022, doi: 10.46947/joaasr412022227.

[14] N. Mangrulkar and K. Singh, “Optimizing Packrat Parsing with Non-Linear Data Structures for Memoization,” in 2023 International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS), IEEE, Oct. 2023, pp. 1–4. doi: 10.1109/ICSSAS57918.2023.10331889.