Program Decomposition and Translation with Static Analysis

Ali Reza Ibrahimzada
alirezai@illinois.edu
University of Illinois Urbana-Champaign
Champaign, IL, USA

ABSTRACT

The rising popularity of Large Language Models (LLMs) has motivated exploring their use in code-related tasks. Code LLMs with more than millions of parameters are trained on a massive amount of code in different Programming Languages (PLs). Such models are used for automating various Software Engineering (SE) tasks using prompt engineering. However, given the very large size of industry-scale project files, a major issue of these LLMs is their limited context window size, motivating the question of *"Can these LLMs process very large files and can we effectively perform prompt engineering?"*. Code translation aims to convert source code from one PL to another. In this work, we assess the effect of method-level program decomposition on context window of LLMs and investigate how this approach can enable translation of very large files which originally could not be done due to out-of-context issue. Our observations from 20 well-known java projects and approximately $60K$ methods suggest that method-level program decomposition significantly improves the limited context window problem of LLMs by 99.5%. Furthermore, our empirical analysis indicate that with method-level decomposition, each input fragment on average only consumes 5% of the context window, leaving more context space for prompt engineering and the output. Finally, we investigate the effectiveness of a Call Graph (CG) approach for translating very large files when doing method-level program decomposition.

ACM Reference Format:
Ali Reza Ibrahimzada. 2024. Program Decomposition and Translation with Static Analysis. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3639478.3641226

1 INTRODUCTION

Machine learning has been widely used for automating various data-related tasks, and software engineering automation is no exception [1, 4, 10–12]. A subset of these machine learning tools called LLMs have shown significant improvements in several code related tasks [3, 5, 14, 15, 19, 20, 23, 26–28]. Prompt engineering constitutes a pivotal element contributing significantly to the achievements of recent LLMs [30]. However, context window, which contains both prompt to the LLM and the response to the prompt from the LLM, is limited. So, prompt crafting, i.e., providing the minimum amount of information to maximize the gain from the LLM response, is crucial. This is not a certain issue for small programs consisting of short methods. However, industry-scale software is typically large and complex, with a lot of dependencies between different components.

This work performs a large-scale study to investigate the effect of method-level program decomposition on the context window of LLMs. More specifically, our early experimental results from 20 well-maintained Apache [6] projects and roughly $60K$ methods demonstrate that real-life industry-scale software is very large and mostly cannot be processed by LLMs, demanding the necessity of fine-grained program decomposition techniques. Moreover, we show that when programs without decomposition fit in the context window of LLMs, they consume the majority of the context, leaving very little space for prompt engineering and the output. In contrast, method-level program decomposition improves the out-of-context issue of LLMs by 99.5% with only consuming 5% of the context window which ultimately enables processing of very large input files. Lastly, we perform a qualitative study on Apache Commons CLI [7] project when translating it using a CG approach to investigate the effectiveness of method-level program decomposition.

2 BACKGROUND

Large Language Models (LLMs) have been extensively used in the domain of Natural Language Processing (NLP) and Programming Language Processing (PLP), achieving state-of-the-art performance in different tasks such as classification [29], translation [21], automated program repair [27], summarization [17]. Prompt engineering involves providing the LLMs with minimal context to enhance their performance on any target task [30]. Recent studies have shown prompt crafting can significantly improve the quality of LLM responses [19, 22, 28]. Given the limited context window of state-of-the-art LLMs, effective decomposition of programs become an essential step when processing very large input files. Existing techniques on program decomposition involve program slicing [24]. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. Each program fragment after slicing is independent and it should guarantee to represent the behavior of the original program. Another class of slicing techniques leverage dependency graph for decomposing very large programs [13]. These techniques mostly use flow graphs (CFG, DFG) for decomposing programs into smaller fragments.

3 APPROACH

In this section, we will discuss our technique for better program decomposition and translation using static analysis. This work performs a large-scale study to answer three main research questions (RQs). **First**, we were interested to understand if industry-scale real-life projects can fit in the context window of widely used LLMs. To

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0502-1/24/04…$15.00
https://doi.org/10.1145/3639478.3641226

Table 1: The effect of method-level program decomposition on a 2K context window model. The analysis has been done on 20 well-known Apache Commons projects.

Project	% Files >2K Tokens	# Methods	Avg. Tokens / Method	% Methods >2K Tokens	% 2K Context
bcel	11.29%	4,094	70.42	0.15%	3.44%
beanutils	29.84%	2,675	107.09	0.07%	5.23%
cli	30.77%	582	97.91	0.17%	4.78%
codec	48.30%	1,788	189.29	0.84%	9.24%
collections	19.34%	6,354	74.37	0.02%	3.63%
csv	27.08%	871	102.53	0.11%	5.01%
daemon	27.78%	60	108.63	0.00%	5.30%
dbcp	38.52%	3,622	63.02	0.03%	3.08%
dbutils	13.54%	869	61.44	0.00%	3.00%
fileupload	16.67%	401	77.8	0.00%	3.80%
geometry	39.13%	6,615	124.93	0.03%	6.10%
imaging	14.78%	2530	143.71	0.20%	7.02%
io	22.07%	5,957	77.94	0.07%	3.81%
jexl	25.70%	3,967	109.37	0.20%	5.34%
lang	40.34%	9,134	103.33	0.12%	5.05%
net	23.83%	2,023	98.22	0.15%	4.80%
pool	22.68%	1,377	94.13	0.00%	4.60%
rng	36.60%	3,245	139.69	0.52%	6.82%
text	28.32%	2,712	99.85	0.04%	4.88%
validator	38.00%	1,181	147.42	0.17%	7.20%
Average	**27.73%**	**3002.85**	**104.55**	**0.14%**	**5.11%**

answer this question, we downloaded 20 well-maintained projects from Apache [6], located each stand-alone `.java` file from these projects and extracted their content. Next, we used the tokenizer of a widely-used open-source LLM called StarCoder [16] to tokenize the content of each file, and measure the number of files which do not fit in a $2K$ context window. We decided to compare all inputs against a $2K$ context size because most recent state-of-the-art LLMs come with a window size of 2048 tokens [2, 8, 16, 18]. **Second**, we wanted to investigate if method-level program decomposition can help with out-of-context problem of LLMs, and if effective prompt engineering is possible when encoding the decomposed method fragments. To address this concern, we performed static analysis on downloaded projects using CodeQL [9] to decompose each class in the projects into method fragments. As mentioned above, we used the same StarCoder tokenizer for tokenizing method fragments and measured the average number of tokens per method and the number of method fragments which do not fit in a $2K$ context window. For effective prompt engineering, we measured the amount of context size each method fragment on average would consume after decomposition. That is, the less space consumed by the input would enhance the performance of the model, leaving more context size for prompt engineering and output tokens. **Third**, we wanted to see if a program decomposition technique can help a simple translation approach in translating project-level programs. We used CodeQL for extracting the CG of each file and created a call dependency graph. Next, we implemented a simple translation technique which leverages the CG and translates the methods in a bottom-up manner. This approach makes sure to translate pure and independent method fragments first, and more dependent ones later by providing contextual information about independent methods.

4 EVALUATION

This section presents the results for our RQs. Our first RQ aims at understanding if industry-scale software can fit in the context window of recent LLMs. Column 2 under Table 1 shows the results for this study. As shown in the table, we observe roughly 30% (min=11.29% and max=48.30%) of input files from real-life projects do not fit in a $2K$ context window model. That is, one-third of files not only fits in the context window, they also do not leave any space for prompt engineering and new output generation. These results indicate an important problem of encoding very large programs by LLMs. To address this issue, we propose method-level program decomposition and discuss our results in RQ2.

Table 2: The effectiveness of method-level decomposition when translating Apache Commons CLI using its Call Graph.

Decomposition Technique	# Source Files	# Out-of-Context Inputs	% Context Occupied
No Decomposition	22	8	36%
Method Decomposition	22	0	3%

Columns 3-6 in Table 1 show the results of RQ2. Motivated by the modularity of large-scale software, i.e., methods tend to be shorter and perform a single function, we decompose each `.java` file in the projects to a set of method fragments in order to address the out-of-context problem of LLMs. On average, each Apache project contains roughly $3,000$ methods under source and test directories, with each method consisting of approximately 100 tokens. Doing such decomposition, we make sure that each fragment is independent and contains a subset of original program behavior as promised in program slicing [24]. As indicated in Table 1, method-level program decomposition improves the out-of-context problem by nearly 99.5% (27.73% down to 0.14%), enabling encoding and processing of very large input files, which otherwise cannot be done due to their huge sizes. Furthermore, our proposed decomposition technique not only addresses the out-of-context issue, it also creates fragments which on average only consume 5% of a $2K$ context window model. This will improve the quality of generated responses as LLMs tend to work well for shorter and well-engineered prompts [25].

In our last RQ, we wanted to investigate if our proposed decomposition technique can be incorporated with an SE task, i.e., code translation, to enable translation of large input files. Table 2 shows a qualitative analysis when translating the source files of Apache Commons CLI [7] with method-level decomposition using its CG. We used an open-source model, i.e, StarCoder [16] for doing the translation. As shown in the table, vanilla translation of each file without any decomposition technique results in out-of-context error of 8 files. In contrast, when using method-level decomposition, all source files can be translated without any problems, by only consuming 3% of context size (a $12x$ improvement compared to no decomposition). Moreover, it is important to note that, our work in this paper focuses on enabling translation of large files, rather than correctly translating them which is a challenging problem by itself. Therefore, we do not validate the correctness of translations.

5 CONCLUSION

Program decomposition of large programs is essential for better prompt engineering of LLMs. In this work, we show that doing method-level program decomposition and CG-based translation enables translating of large input files. As part of our future work, we will explore how our approach can be combined with other techniques such as slicing and dependency graph decomposition.

REFERENCES

[1] Open AI. 2023. Open AI ChatGPT. https://openai.com/blog/chatgpt

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[3] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. *arXiv preprint arXiv:2310.09342* (2023).

[4] Cursor. 2023. Cursor Code Editor. https://cursor.sh/

[5] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[6] The Apache Software Foundation. 2023. Apache. https://github.com/apache

[7] The Apache Software Foundation. 2023. Apache Commons CLI. https://github.com/apache/commons-cli

[8] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[9] GitHub. 2023. CodeQL. https://codeql.github.com/

[10] GitHub. 2023. GitHub Copilot. https://github.com/features/copilot

[11] Google. 2023. Google Bard. https://bard.google.com/

[12] Google. 2023. Google PaLM. https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html

[13] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 35–46. https://doi.org/10.1145/53990.53994

[14] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2023. Automated Bug Generation in the era of Large Language Models. *arXiv preprint arXiv:2310.02407* (2023).

[15] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. 2022. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 70–81.

[16] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[17] Yang Liu. 2019. Fine-tune BERT for extractive summarization. *arXiv preprint arXiv:1903.10318* (2019).

[18] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[19] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. *arXiv preprint arXiv:2308.03109* (2024).

[20] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2023. Towards Causal Deep Learning for Vulnerability Detection. *arXiv preprint arXiv:2310.07958* (2023).

[21] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[22] Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. 2023. LeTI: Learning to Generate from Textual Interactions. *arXiv preprint arXiv:2305.10314* (2023).

[23] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. *arXiv preprint arXiv:2309.00608* (2023).

[24] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[25] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[26] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[27] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[28] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[29] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems* 32 (2019).

[30] Qinyuan Ye, Maxamed Axmed, Reid Pryzant, and Fereshte Khani. 2023. Prompt Engineering a Prompt Engineer. arXiv:2311.05661 [cs.CL]