Noncommittal Commits: Predicting Performance Slowdowns in Version Control History

Daniel Nichols[†], Dilan Gunawardana[†], Aniruddha Marathe[*], Todd Gamblin[*], Abhinav Bhatele[†]

[†]Department of Computer Science, University of Maryland, College Park, MD 20742 USA
[*]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA

***Abstract*—Scientific software in high performance computing is becoming increasingly complex both in terms of its size and the number of external dependencies. Correctness and performance issues can become more challenging in actively developed software with increasing complexity. This leads to software developers having to spend larger portions of their time on debugging, optimizing, and maintaining code. Making software optimization and maintenance easier for developers is paramount to accelerating the rate of scientific progress. Fortunately, there is a wealth of data on scientific coding practices available implicitly via version control histories. These contain the state of a code at each stage throughout its development via commit snapshots. Commit snapshots provide dynamic insight into the software development process that static analyses of release tarballs do not. In this abstract we propose a new machine learning based approach for studying the *performance* of source code across code modifications. First, we present a novel methodology for preparing time-series performance data that preserves syntactic structure. Then, we train a neural network to predict if a commit introduces any code changes that will likely impact performance negatively. We validate the trained model on performance data from scientific code repositories.**

***Index Terms*—version control history, source code changes, performance degradation, machine learning model**

Modern codebases of computational science and engineering software are becoming increasingly complex in their design and scale. Production scientific software can have hundreds of thousands of lines of code in addition to numerous external dependencies. Maintaining these software stacks imposes a significant burden on developers. Small changes can have drastic impacts on organization, correctness, security, and/or performance. In particular, performance of a code can change significantly across a codebase's version control history. Within a large, complex codebase, it becomes difficult for developers to predict the performance impacts of code changes or assign culpability to observed impacts.

Commonly followed practice in software engineering has centered around addressing issues related to correctness and performance changes across commit snapshots. For instance, unit and regression tests are used to help identify correctness and performance breaking commits. These tests, like many other detection mechanisms, are incomplete as they are designed by software engineers and prone to the same errors as the original code. Additionally, they need to be hand-designed, and need to be executed periodically (after every commit or nightly) which can be time consuming and expensive.

With the recent explosion of machine learning (ML) research, many works have focused on supplementing human-designed software engineering tools with ML-based assistance. These works are particularly timely as a wealth of software development data exists online through public facing version control repositories. Most of this research has focused on developer productivity, correctness, and security [1]–[13].

Applying ML-assisted tools for code to studying performance, however, poses many new challenges. Engineering a meaningful dataset to learn performance metrics from code structure is not straight forward. First, collecting performance data involves building a code and its dependencies, and then executing it with representative input problems. To collect large amounts of data, this process needs to be automated. However, build processes vary significantly across code repositories and can even change across commits within a single repository. Second, performance is often dependent on more than just code structure. For example, specific code changes may degrade performance on one instruction set architecture, but not another. Finally, code changes may only affect certain code paths, which means dynamic runtime information about control flow needs to be used to prune the data on code changes. This makes it difficult to collect large amounts of performance data for an arbitrary code, which, in turn, makes training deep neural networks more difficult as they need large amounts of data.

In this abstract we tackle the data collection challenges mentioned above by developing a novel methodology for preparing time-series performance data that preserves syntactic structure. We develop an end-to-end system for collecting meaningful performance data across an application's version control history. We use the processed data to train a neural network to predict if a commit introduces any code changes that will likely impact performance negatively. Additionally, we demonstrate the effectiveness of this methodology on scientific code repositories.

Data Collection and Augmentation

To gather performance data on scientific applications, we choose to collect data on Kripke [14] and Laghos [15]. These are two proxy applications that mimic DOE production codes. This data includes source code and performance results for each commit. The performance results are collected by building and running the commit on the Quartz system at Lawrence Livermore National Laboratory.

To study the code structure we generate Abstract Syntax Trees (ASTs) for the source code at each commit. Several compilers provide functionality to view their generated ASTs during compilation. We use the clang c++ compiler and its `-ast-dump` flag to generate ASTs from c++ source files.

When predicting relative performance we could use the entire AST as input into the model, however, this may include inactive regions of code. Some AST changes may have significant potential impact on performance, but not lie in the execution path for a particular input. These portions of the AST may give misleading information to the model during training and should be pruned out.

To accomplish this we assign run times to branches of the AST using a profiled CCT. This is done using the file and line numbers provided by both trees. We use HPCToolkit [16] and Hatchet [17] to collect and process CCTs. Branches of the AST which execute below some threshold can be pruned out. We remove portions whose inclusive runtime is less than 1% of the total runtime. This also removes insignificant code paths even if they do execute. These are less interesting and can be removed to reduce the size of the feature space.

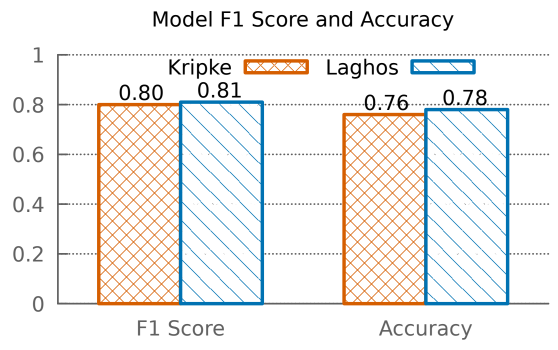

Fig. 1. The F_1 score and accuracy of the LSTM on each testing data set. Both models train to similar results with the model performing better for the Laghos data set.

To generate a more structured data format from the ASTs we use the GumTree algorithm [18] to compute edit scripts. When applied in order, the computed actions in the edit script map one AST to another. These are not necessarily unique and, for the GumTree algorithm, optimal, however, they are often small and intuitive.

Due to the limited number of commits in these repositories and the demanding requirements of many DL models we need a way to increase the number of samples in the data set. We accomplish this in three different ways: making use of the symmetry in the data points, partitioning the ASTs into code regions, and expanding the diffs beyond a single commit.

Since each data sample represents the relative code and performance changes between hash h_i and h_{i+1}, then we can swap the input features and output label to create a new valid data point. This increases the size of the data set by $2\times$.

Each AST covers the entirety of code within a commit, however, it can be split into code regions to gain performance data at a finer granularity. The code can be partitioned into any granularity up to a single line. In this work we split each function into its own sample. This will increase the number of data points by a multiple of the average number of functions edited in each commit.

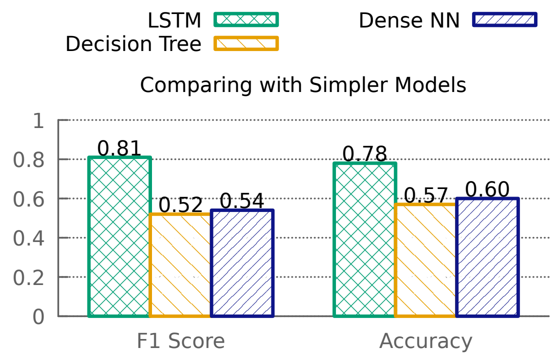

Fig. 2. The F_1 score and accuracies of the different ML models train on the Laghos data set. Only the LSTM achieves reasonable results, which emphasizes the necessity of preserving the entire edit script.

We can also consider code changes across multiple commits as input samples. This can be done with a sliding window over the existing samples. To add w samples we compute the diffs for (h_i, h_{i+1}) to (h_i, h_{i+w}) and use the absolute times to get the relative performance and assign the output label. For a data set of size N this will increase the number of samples by $w\times$ up to N^2 total samples. We empirically determined the best window size for our data set to be $w = 8$.

Edit scripts are sequential and variable in length, so predicting relative performance based on them is a sequence classification problem. A common ML model for this type of task are Long Short Term Memory (LSTM) neural networks. The network architecture used in this work is

$$\text{LSTM} \rightarrow \text{Dropout} \rightarrow \text{Dense} \rightarrow \text{Output}$$

The beginning is 2 LSTM layers with 256 hidden units each. After the LSTM layers is a dropout layer with a 0.2 dropout rate. Finally, a traditional dense layer feeds into a log-loss function. We employ the Adam optimizer to fit the network's parameters.

After training the model achieves an F_1 score of 0.8 and 0.81 on Kripke and Laghos, respectively. Similarly it classifies their relative performance with 76% and 78% accuracy (see Figure 1). When compared with non-sequential classification using summary statistics of the edit scripts the LSTM largely outperforms the simpler models (see Figure 2). This shows the representative capacity of the LSTM model.

In summary, we have presented a methodology for overcoming the barriers in performance data collection across version control history. We then used this data to show how machine learning can be employed to predict performance degradation based on code changes in a repository.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-838436).

References

[1] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 60–70, 2018.

[2] Q. Zhu, Z. Sun, Y. an Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[3] M. Wu, N. D. Goodman, C. Piech, and C. Finn, "Prototransformer: A meta-learning approach to providing student feedback," *ArXiv*, vol. abs/2107.14035, 2021.

[4] N. Pinnow, T. Ramadan, T. Z. Islam, C. Phelps, and J. J. Thiagarajan, "Comparative code structure analysis using deep learning for performance prediction," 2021.

[5] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Deep similarity learning-based type inference for python," *ArXiv*, vol. abs/2101.04470, 2021.

[6] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 344–354.

[7] J. Huang, D. Tang, L. Shou, M. Gong (YIMING), K. Xu, D. Jiang, M. Zhou, and N. Duan, "Cosqa: 20,000+ web queries for code search and question answering," in *ACL-IJCNLP 2021*, May 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cosqa-20-000-web-queries-for-code-search-and-question-answering/

[8] J. Senanayake, H. Kalutarage, and M. O. Al-Kadri, "Android mobile malware detection using machine learning: A systematic review," *Electronics*, vol. 10, no. 13, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/13/1606

[9] R. Gupta, A. Kanade, and S. Shevade, "Neural attribution for semantic bug-localization in student programs," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[10] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 2244–2253. [Online]. Available: https://proceedings.mlr.press/v139/cummins21a.html

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[12] V. Nitin, A. Saieva, B. Ray, and G. E. Kaiser, "Direct : A transformer-based model for decompiled identifier renaming," in *NLP4PROG*, 2021.

[13] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[14] A. Kunen, T. Bailey, and P. Brown, "KRIPKE-a massively parallel transport mini-app," *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2015.

[15] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012. [Online]. Available: https://doi.org/10.1137/120864672

[16] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[17] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Nov. 2019, lLNL-CONF-772402. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356219

[18] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982