RAX: A porting complexity evaluation tool for RISC-V

Anonymous Author(s)

ABSTRACT

In recent years, RISC-V instruction set architecture(ISA) has attracted the attention of the information technology industry due to its open source, modular design, and simplicity. However, there is a lack of standardized metrics and methods to measure software architecture portability, which helps to establish a mature software ecosystem in RISC-V. A theoretical framework is urgently needed as a guide to summarize important porting-related factors and design an automated RISC-V-oriented porting complexity evaluation tool to solve the problem. We propose RAX, it will help recommend software with varying degrees of porting complexity to development teams. RAX first retrieves porting modification information of the adapted project according to the Git version control tool, classifies it through manual analysis, and selects the most important porting factors to design the code positioning mode. Additionally, it incorporates Cyclomatic Complexity, which is from code evaluation domain, to enhance assessment accuracy and measure the overall complexity of the software. We establish a high-quality data set of software porting complexity, and select random forest machine learning model as the final classifier through multi-model comparison.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; *Software post-development issues.*

KEYWORDS

ISA(Instruction set architecture), RISC-V, Porting Complexity Evaluation

1 INTRODUCTION

RISC-V is an emerging open-source and reduced ISA which avoids potential risks such as intellectual property issues [18]. Compared to traditional architectures, RISC-V has a simpler design, which lowers R&D threshold. Its modularization and scalability allows developers to customize the architecture flexibly to adapt to different application scenarios [17].

Although RISC-V has already made initial progress in having a relatively complete tool chain, it still requires the support of a large number of application software [2]. However, establishing a mature software ecosystem still face challenges. Currently, there is a lack of standardized metrics to measure software architecture portability,and porting planning relies on domain expertise. Du Jiman et al. have developed a basic statistical evaluation plan for CPU architecture code in project [7], however, their work lacks a thorough validity analysis and comparison of factors,and fails to consider the complexity associated with the software operation and maintenance process. It is difficult for developers to estimate the complexity of the porting project and accurately allocate the necessary resources,and there are also challenges related to variations in task difficulty and developer proficiency. In order to solve this problem,one of the methods is to make analogies with adapted projects to gain porting experience,but it is difficult to extract factors by manual sorting;Even if developers have sensitivity to architecture-related modules, it is still challenging to accurately obtain the workload. RAX can provide developers with accurate evaluation, recommend software with different levels of porting difficulty based on their skill levels. The contributions of this paper are as follows:

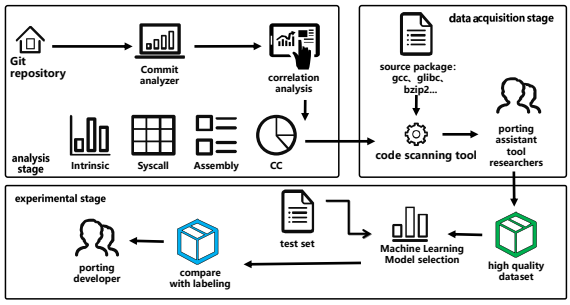

Figure 1: Flow chart for method to build RAX

(1) Firstly, we conducted analysis on selection of factors related the porting evaluation,investigated the significance of code complexity metrics in the context of porting work and incorporated Cyclomatic Complexity as a factor to assess the overall complexity of the project.
(2) Based on porting-related factors, we designed a scanning tool to collect porting complexity vectors and ultimately selected random forest machine learning model as the classifier for RAX.
(3) Evaluate the effectiveness of RAX. We expanded the dataset by nearly double and verified the effectiveness of 100 expanded projects through their respective communities or forums. We received 50 responses, out of which 30 were considered valid responses. In comparison to Du Jiman's tool,RAX demonstrated improvements.

2 RAX

The following is an introduction to the process of RAX construction, as shown in Figure 1. During the analysis phase,architecture-related factors are obtained through the Commit tool for selection. A scanning tool is built to get potential modification workload. Finally,an appropriate machine learning model is selected as a classification.

2.1 Motivation

The section provides a validity analysis of porting-related factors on RAX. First,we use code complexity metrics to evaluate porting work. The software metrics of code Cyclomatic Complexity(CC) can help developers make macro judgments on software complexity

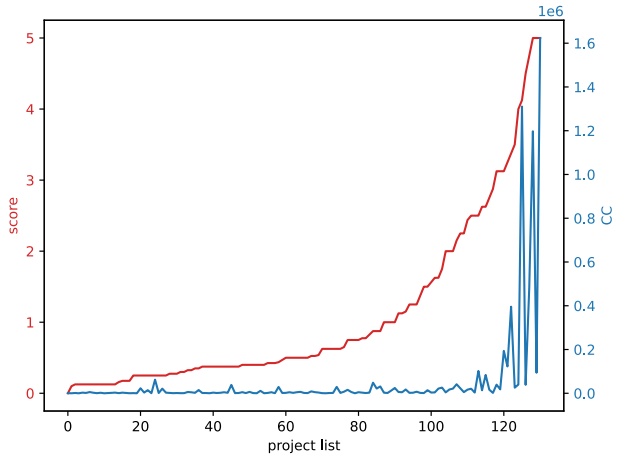

Figure 2: The relationship between score and CC

and maintenance difficulty, evaluate potential and hidden porting difficulties, and help the tool improve accuracy [6]. At present, CC indicators are mostly used in project code quality assessment, defect inspection, code reconstruction, etc [8]. During the porting process, developers will target a large number of project source codes, scripts, assembly codes,and a macro understanding of the project is required in the early stage of porting. This part of the work has a lot of overlap with the scope of code complexity.

2.1.1 Factor 1: Cyclomatic Complex(CC). RAX tool uses the lizard CC detection tool and optimize its functions,we will introduce it in Section 2.3. This study analyzes the relationship between CC and software porting complexity using the Spearman Rank correlation coefficient analysis method. We invited five Master's students who are engaged in research on porting auxiliary tools to score the porting complexity(score) of 131 software adapted for RISC-V,collected from the OpenEule list and GitHub. According to the collection of Internet events, log analysis, source code dismantling, and similar project analogy [11], we form a scoring interval in the range of [0 5]. We then calculated the Spearman correlation coefficient and conducted significance analysis between score and the CC factors. The Spearman coefficient is 0.608, with a P-value of 1.39e-14, which is less than 0.01. This indicates a significant correlation between the two variables. The results suggest that CC has a moderate positive correlation with porting difficulty. This is because developers need a holistic understanding of the project during the porting process, and CC can represent the overall complexity of the project, potentially reflecting porting barriers, as shown in Figure 2. Secondly,we focus on the code associated with architecture in the project,with the goal of predicting porting workload through these code.

2.1.2 Factor2:Arch_code. First, extract all possible architecture-related factors. Use the Git tool to split Commit into different files using git-diff and filter out the surrounding 20 lines of content containing architecture keywords for detailed analysis of the modifications. The manually analyzed Commit generally includes the following parts: first, functional supplements, including lack of dependent package calls, performance optimization and test cases

Table 1: The Spearman correlation coefficient of score and architecture-related factors

Type	Score
conditional compilation	0.73
Builtin	0.61
Intrinstic	0.74
syscall	0.69
asm	0.66
build_scripts	0.43
time	-0.16
authors	0.43
lines of arch	0.16-0.40
files of arch	0.34-0.41
keyword count	0.17-0.43

for architecture porting;second, detail adjustments for data type and Build script;finally, key function implementation, assembly code, built-in function, system call, etc. Among these factors, we select more objective ones that can be automatically located through regular expression matching.

Arch_code defines assembly code, conditional compilation and macro structure, Intrinsic functions, Builtin functions, architecture-related build scripts, and system calls. We use an automated architecture binding code scanning module to obtain statistics on the number of lines and frequencies of Arch_code.

Continue to collect other factors that may affect the evaluation of porting complexity,and select the following parameters through the Commit analyzer:architecture-related code lines, architecture-related files, and architecture keywords. The number of architecture-related code lines and files is located using the keywords of X86, ARM, MIPS, RISCV, SPARC and other architectures. Correlation comparison between Arch_code and other factors acquired by Commit analyzer,as shown in Table 1:

The results indicate that Arch_code has a higher correlation with score compared to other factors, and it also aligns with the significant results of the Spearman correlation test. The use of only architecture keywords for statistics yields moderate effectiveness in evaluating tool because keywords can only be detected from comments and macro sections of Commit, while critical porting code may lack these keywords,but the keyword count can reflect the amount of porting modification from the side. As shown in the figure 2, there may be great ambiguity in keyword positioning using different architectures. So we only selected x86 for positioning, which represents the most fair workload. In the end, the parameters of Arch_code show a correlation of 0.43 to 0.75 with score, indicating a strong relationship. However, using a single factor alone does not result in a significantly strong correlation, as the combined modifications of these factors make up the porting workload.

To design RAX that assists developers with porting work,the following research questions are proposed:

Research Question 1: Developers often struggle to make fine-grained differentiations in the complexity of porting projects. How can we select architecture-relevant factors to design an automated tool that assesses the porting difficulty? We analyzes the porting

modifications obtained from developer Commit in the Git tool. We conducts an effectiveness analysis of the selected factors and filters out strongly correlated ones that can be automatically identified using the Lines of Code (LOC) approach. Additionally, we propose a CC assessment method to jointly establish a data set for porting complexity.

Research question 2: How to select a classifier to obtain effective classification results for the tool? Using the processing method of unbalanced datasets, multiple machine learning models are compared to select the final classifier.

2.2 Dateset

We used the project upstream warehouse to build the data set,that projects mainly targeting C and C++ are from the OpenEuler:Stage2 list and GitHub,and invited 5 postgraduate students who are engaged in the research of RISC-V porting auxiliary tools to participate in the experimental evaluation. Specific evaluation methods can be found in Section 2.1.1. The evaluation scores are divided into three categories: low, medium, and high. In the end, RAX collected a total of 134 projects.

2.3 Method

The RAX combines the CC calculation module and the Arch_code location module to obtain the porting vector,and the complexity vector and score together form the data set.

2.4 Lizard CC calculation module

Lizard's default computing method is basically consistent with McCabe's theory [12]. However, Lizard tool only detects defects at the function level and uses the average function CC for file-level statistics, which is not suitable for current porting complexity detection requirements [5]. This paper proposes a new method for evaluating CC specifically for porting assessment work:

(1) First, calculate the function-level CC for all files in the project using McCabe's method.
(2) Following Liu Huihui's mention of System Cyclomatic Complexity (SCC) in software evolution assessment techniques based on code change detection [10], we define SCC as the total sum of function-level values in the system. The advantage is that it can avoid the use of overly complex methods, which will cause deviations to affect the classification results. Additionally, it eliminates the problem of decreased detection performance caused by program transformation into syntax abstract trees and control flow graphs.

2.5 Arch_code location module

we form the Arch_code dictionary:

(1) Conditional compilation and architecture macros are used as preprocessor macros to enable projects to select execution on different hardware architectures, thus demonstrating the additional workload of architecture porting. Use conditional compilation statements, such as "if defined", "ifdef", in conjunction with all possible architectur macros,"__x86_64__". We use a similar bracket matching algorithm to cover all the code within macros, avoiding the impact of nested layers. Additionally, we only collect the keyword dictionary for x86 architecture. This is because different architectures may reflect varying workloads and difficulties, while considering that multiple architectures blurs the classification performance in practical work.
(2) Assembly code Leverages the advantages of architecture-specific instructions to enhance performance. Considering that architecture changes inevitably require consideration of the assembly section, we use matching forms such as "__asm__" and "__volatile__". In future work,we will consider striving to achieve the currently unachievable architecture identification function of assembly code and identify assembly code in the form of instructions;In actual work, the assembly code of different architectures may represent different levels of difficulty. For example, the assembly code of ARM is relatively simple. At the same time,there are also situations where some assembly codes only need to serve a specific architecture and do not need to be fully modified when porting to RISC-V. This is why we cannot achieve Fine-grained evaluation,we simply count the number of lines.
(3) Intrinsic functions are interfaces provided by compilers that are closely related to the architecture. Calling Intrinsic functions can replace assembly code. We compile dictionaries of Intrinsic functions based on official documentation from x86. Builtin function also employ a similar approach;In the future,we may assign difficulty weights to them. Currently,we only perform frequency statistics on them. Additionally,we will continue to supplement the analysis of whether the functions already have corresponding implementation for RISC-V and whether they are truly needed in RISC-V.
(4) System calls are encapsulated by the operating system kernel to provide hardware calling. Considering glibc's support for system call encapsulation, reorganize the sys-call dictionary. The system calls that do not support RISC-V have been filtered out currently.
(5) In the porting process, adjustment of build scripts is necessary to ensure smooth compilation. We locate architecture keywords in the build scripts. In the future, architecture-related factors will consider even finer-grained classification. For instance,considering the impact of existing porting auxiliary tools, including intrinsic translation tools and assembly translation tools,on porting complexity assessment.

3 RESULT AND ANALYSIS

To answer research question 2, based on the work of Chen Xiang et al. [9], aiming at the status quo of unbalanced datasets and small sample datasets, we try to use ensemble learning methods [16], SMOTE [13], and random oversampling methods for data preprocessing. Ultimately, we chose to train four models: Random Forest, Support Vector Machine, Adaboost, and Xgboost. By comparing multiple indicators, we select the Random Forest machine learning as the classifier.

We continued to expand 100 projects as a supplementary data set, of which 87 were able to predict correctly. The prediction results were determined through comments from the community

Table 2: The Spearman correlation coefficient of score and architecture-related factors

	Accuracy	Pmacro	Rmacro	F1macro
Tool 1	0.838	0.802	0.880	0.823
RAX	0.857	0.817	0.873	0.836

Table 3: Effectiveness comparison chart

	Tool 1		RAX	
	TP	FN	TP	FN
Low	15	3	17	1
Medium	5	2	5	2
High	3	4	7	0
Sum	23	9	29	3
project	32		32	

and forums. Appendix presents a comparative assessment of the porting complexity evaluation tool developed by Du Jiman et al. across various projects.

After adding the code complexity factor and modifying the original tool design model, RAX's result prediction accuracy is improved. This is because code complexity can effectively distinguish projects with low porting difficulty from projects with medium and high porting difficulty. Motivation Chapters present the cause analysis. Adding factors also makes RAX more robust and more friendly to projects with unbalanced complexity vectors. For example, kernel-level projects, boot layers, hardware drivers and other projects may involve a large amount of assembly code,corresponding to larger porting complexity,but other factors of the original tool design are at low values,which is not conducive to distinguishing complexity. Chibios,u-boot are facing such a situation. Within this assessment, there are instances of inaccurate predictions. An analysis of these discrepancies reveals that certain projects themselves are of substantial complexity,containing a significant amount of architecture-related code defined by Arch_code. However,in the actual porting process, there is support from various porting auxiliary tools as well as dependency packages adapted RISC-V,like the assembly code used for acceleration in the portmidi project has a simple C- equivalent available. As a result,the expert assessors did not assign high difficulty scores to these projects. At the same time,it is also due to the existence of a situation where the project is large but there is less Arch_code,and adding CC cannot avoid it,like cgal project. Sometimes developers design their work has generic code for unknown architecture and CPU architecture parts are well separated from the common code,that just need only to adjust the macro to recognize RISC-V,like nuttx.

4 RELATED WORK

The development of the RISC-V ecosystem is rapidly progressing. In the realm of scientific computing, Tine Blaise et al. [15] accomplished the porting of the OpenCL framework to RISC-V through a compiler framework, thereby expanding RISC-V's accessibility to a wider range of scientific computing applications. In the context of architectural migration of foundational software and libraries, Cheng Yuanhu et al. [14] designed an embedded multi-instruction set processor,addressing the software ecosystem challenges present in the RISC-V embedded domain. As a result, auxiliary efforts in porting are highlighted as valuable endeavors. This tool approaches code complexity from the perspective of its evaluation role in software evolution and maintenance,considering its influence on porting complexity assessment [3]. Alvares et al. [4] proposed using the development team's "tolerance" for code complexity to infer team development capabilities and performance indicators for projects. Vard et al. [1] emphasized that the internal quality of software impacts developers' capabilities and maintenance durations. They conducted a comprehensive study on various existing code complexity metrics' validity and suggested incorporating empirically observed code features as complexity triggers.

5 CONCLUSION

We propose RAX, with the aim of recommending projects of varying porting difficulties to developers. We incorporate the code complexity to reflect developers' actual workload. Additionally, we conduct a correlation analysis of Arch_code; Analyze the defects and improvements of factor statistical methods, and finally build a scanning tool. Using a Random Forest machine learning model, we perform classification learning on 234 projects collected from the OpenEuler:Stage2 list and GitHub. When compared to the previous version of the tool, our approach yields improved predictive performance. Moreover, RAX provides developers with the functionality to locate target code segments. Our next steps involve attempting text classification methods to categorize the text results of the scanning tool, designing more granular statistical methods for architecture-related factors.

REFERENCES

[1] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* (2017).

[2] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: a device for secure transaction approval. In *Symposium on Operating Systems Principles*.

[3] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. 1993. Software Complexity and Maintenance Costs. *Commun. ACM* 36, 11 (1993), 81–94.

[4] Marcos Alvares Barbosa, Fernando Buarque de Lima Neto, and Tshilidzi Marwala. 2016. Tolerance to complexity: Measuring capacity of development teams to handle source code complexity. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 002954–002959. https://doi.org/10.1109/SMC.2016.7844689

[5] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abugharah, and Lucas Bang. 2021. Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 29–32. https://doi.org/10.1109/ICSE-Companion52605.2021.00028

[6] Andrea Capiluppi, Alvaro Faria, and J. F Ramil. 2005. Exploring the relationship between cumulative change and complexity in an open source system evolution. In *IEEE*.

[7] jiman Du. 2023. DESIGN AND IMPLEMENTATION OF A RISC-V PORTING TASK PLANNING SYSTEM BASED ON RPM SOURCE PACKAGE DEPENDENCY. (2023).

[8] G. K Gill and C. F Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *Software Engineering IEEE Transactions on* 17, 12 (1991), 1284–1288.

[9] Haibo He and Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering* 21, 9 (2009), 1263–1284. https://doi.org/10.1109/TKDE.2008.239

[10] LIU Huihui. [n.d.]. *Techniques of Evaluating Software Evolution Based on Code Change Detection.* Ph.D. Dissertation. Southeast University.

[11] Wu JZ Liang GY, Wu YJ and Zhao C. 2020. Open Source Software Supply Chain for Reliability Assurance of Operating Systems. *Journal of Software* 31, 10 (2020), 18.

[12] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[13] Lourdes Pelayo and Scott Dick. 2007. Applying Novel Resampling Strategies To Software Defect Prediction. In *NAFIPS 2007 - 2007 Annual Meeting of the North American Fuzzy Information Processing Society*. 69–72. https://doi.org/10.1109/NAFIPS.2007.383813

[14] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, and Luca Benini. 2017. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.

[15] Blaise Tine, Seyong Lee, Jeffrey S. Vetter, and Hyesoon Kim. 2021. Bringing OpenCL to Commodity RISC-V CPUs. (6 2021). https://www.osti.gov/biblio/1830102

[16] Shuo Wang and Xin Yao. 2013. Using Class Imbalance Learning for Software Defect Prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443. https://doi.org/10.1109/TR.2013.2259203

[17] Xi Wang, Antonino Tumeo, John D. Leidel, Jie Li, and Yong Chen. 2019. MAC: Memory Access Coalescer for 3D-Stacked Memory. (9 2019). https://doi.org/10.1145/3337821.3337867

[18] A Waterman, Y Lee, Da Patterson, and K Asanovi. 2014. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. (2014).