RAX:Efficiently Evaluating the Complexity of Porting for RISC-V

Anonymous Author(s)

ABSTRACT

In recent years, the open-source and the advantages of reduced ISA of the RISC-V architecture are attracting active promotion of porting within the industry.Due to the lack of public available standards for assessing software portability, the software porting process for the RISC-V architecture has been predominantly reliant on expert experience, which is inefficient and time consuming. RAX is a automated tool which aids the C/C++ programmer in evaluating complexity of software during porting for RISC-V.RAX provides precise evaluation solutions and builds a high-quality dataset for porting complexity,enabling recommendations of suitable software with difficulty levels for developers of varying skill levels.We evaluate RAX on 72 popular and typical real-scenario C/C++ projects. The experimental result shows that 51 of them have been confirmed by developers within community.Interested users can explore the open-source implementation of RAX at https://github.com/wangyuliu/RAX-2024, or watch the RAX video demo at https://www.youtube.com/watch?v=SLphW3Ra4WM.

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 Instruction Set Architecture (ISA) has garnered significant attention from academia, and particularly the industry recently, due to its open-source and reduced design[26].Many distribution manufacturers are actively promoting the porting of the RISC-V, for example,OpenEuler has joined RISC-V International organization, and RISC-V has officially become openEuler's official support architecture.The main repository of openEuler 23.03 has successfully ported 4,244 software packages to the RISCV64,while the openEuler 23.09 main repository has currently completed the porting work of 5,694 software packages[2].OpenCloudOS Kernel Stream 2207.2 kernel version adds support for the RISC-V 64 architecture[1].

Despite the rapid development of the RISC-V ISA, which foundational software such as operating systems and compilers are relatively mature, there is still a significant demand for porting a large number of application software[8].To adapt to the RISC-V,efficiently porting is greatly depending on prior knowledge of experts, and the lack of standardized evaluation criteria for software portability can lead to unreasonable development planning.Therefore, it is desirable to propose a method that automates the evaluation of porting complexity.This will enable different development teams to access software packages that are tailored to their abilities, thus assisting distribution vendors in the porting of RISC-V architecture.

Many related works focus on the code complexity and portability evaluation,but studies for CPU architectures are currently lacking in these areas[17],[23],[22].Du et al. have developed a basic statistical evaluation plan for CPU architecture code[16].However,their work lacks a thorough validity analysis of factors,and may not consider the complexity factors associated with the software operation and maintenance process. To overcome these challenges, we propose RAX, an effective porting complexity evaluation tool which accurately locates the CPU architecture code and calculates the code complexity of the software.The contributions of this paper are as follows:

- We analysis the actual porting modifications and figure out one of the most important factors,Cyclomatic Complexity(CC), exploring its significant value in porting scenarios.
- Based on architecture-related factors, we propose RAX, to efficiently assess the complexity of software during porting to the RISC-V.
- We build a dataset of real-world C/C++ projects and use it to evaluate the effectiveness of RAX.Our findings indicate that RAX outperforms other tool, delivering substantial improvements in terms of accuracy.

2 MOTIVATION

To help developers more accurately assess the complexity of C/C++ projects, we aim to answer the following research questions:

RQ 1: How can we identify architecture-relevant important factors affecting porting efficiency?

RQ 2: How to implement fine-grained and automated tools to help developers accurately assess the complexity of their projects in real world. How to solve the problem of fuzzy classification of projects with medium and high porting complexity in existing tools? As we all known, the metrics of projects complexity is influenced by many factors, such as XXX,XXX. To answer RQ 1, we introduce the code Cyclomatic Complexity(CC) and Arch_code.

To answer RQ 1, First, we try to 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 and maintenance difficulty. By evaluating potential and hidden porting difficulties,code complexity helps the tool improve accuracy [12]. At present, CC indicators are mostly used in evaluating code quality, inspecting defect, reconstructing code, etc [13]. During the porting process, developers will target a large number of project source codes,build 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.Through our survey of community, we discovered that even though architectural code can be separated quite clearly from common code, when faced with large-scale projects, we still require numerous iterative tests to uncover unknown architectural code. Building upon this, we study the quantification of the significance of cyclomatic complexity in porting work.

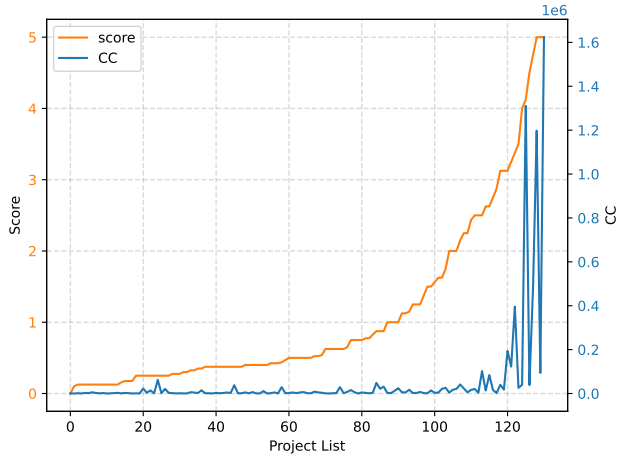

Figure 1: The relationship between score and CC

2.1 Factor 1: Cyclomatic Complex(CC)

We use the lizard CC detection tool and optimize its functions,and it will introduce it in Section 3.1. this section calculate the correlation coefficient using the Spearman Rank. We invited five Master's students who are engaged in research on porting auxiliary tools to score the porting complexity(score) of 131 projects adapted for RISC-V,collected from the OpenEuler list[3] and GitHub. According to the collection of Internet events, log analysis, source code dismantling, and similar project analogy [18], we form a scoring interval in the range of $[0, 5]$. The Spearman coefficient of CC and score is 0.608, with a P-value of 1.39e-14, which is less than 0.01. 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, the relationship of factors shown in Figure 1.

Second, we consider the effectiveness of the architecture-related code structure proposed by other tool and conduct comparative experiments using more potential factors.

2.2 Factor2:Arch_code

We select factors with the greatest potential from existing tool and define Arch_code. Arch_code includes assembly code, conditional compilation and macro structure(CCM), Intrinsic functions, Builtin functions, architecture-related build scripts, and system calls. We use an automated tool to obtain statistics of lines and frequencies of Arch_code.

We further collect other factors that may reflect the porting complexity,and select the following parameters through the Commit analyzer:architecture-related code lines, architecture-related files, and architecture keywords.architecture-related code lines and files is located using the keywords of the RISC-V architecture, such as riscv, rv32, rv64, RISC-V, RISCV and so on.

Figure 2 shows the correlation comparison between Arch_code,CC and other factors acquired by Commit analyzer: The results indicate that Arch_code and CC has a higher correlation with score compared to other factors, and we use Peason correlation method to build the heat map. The use of only architecture keywords for statistics yields moderate effectiveness because keywords can only be detected from comments and macro sections of Git commits, while critical porting code may lack these keywords,but the number of keywords can reflect the amount of porting modification from the side. Arch_code show a correlation of 0.45 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.

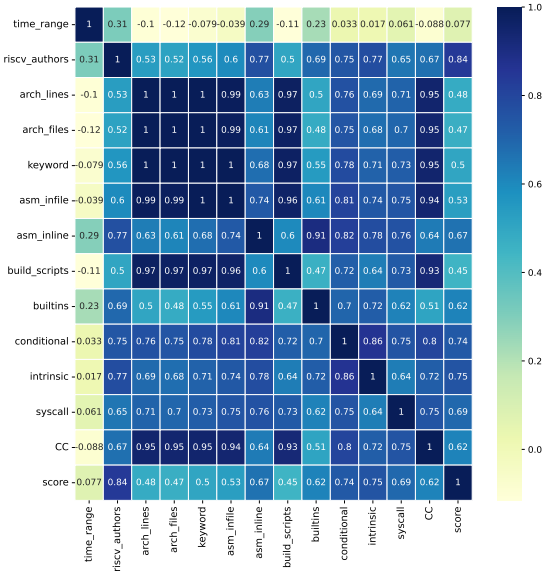

Figure 2: The heat map of pearson correlation

In the end,our tool selection was CC and the Arch_code build scanning tool,as shown in Table 1.

3 RAX

The following is an introduction to the process of RAX construction, as shown in Figure 3. During the analysis phase,architecture-related factors are obtained through the Commit tool for selection while investigating the impact of code complexity metrics on porting. A scanning tool is built to get potential modification workload. Finally,an appropriate machine learning model is selected as a classification.

3.1 Implementation

RAX obtains porting complexity vector through the CC calculation method and Arch_code statistical value.

3.1.1 Lizard CC calculation scan function. Lizard's default computing method is basically consistent with McCabe's theory [19]. However, Lizard tool only defects at the function level and uses the average function CC for file-level statistics, which is not suitable

Table 1: Tool factors comparison

	Full Assembly	Inline Assembly	Intrinsic	Builtin	System Call	CCM	Buildscripts	CC
Other tool	✔	✔	✔	✔	✔	✔	✔	✗
RAX	✔	✔	✔	✔	✔	✔	✔	✔

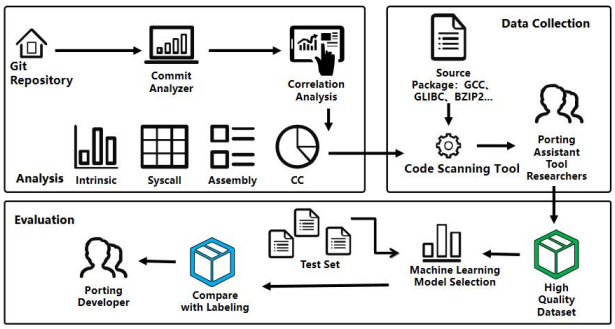

Figure 3: Flow chart for method to build RAX

for current porting complexity detection requirements [11]. This paper proposes a new method:

First, we calculate the function-level CC for all files in the project using McCabe's method.Following Liu's mention of System Cyclomatic Complexity (SCC) in software evolution assessment techniques based on code change detection [15], we define SCC as the total sum of function-level values in the system. The advantage is that it can avoid the use of complex methods, which will cause deviations to affect the classification results. Additionally,due to the system's complexity being calculated based on a single function,it eliminates the problem of decreased detection performance caused by the conversion of such a large code structure of all modules into into abstract syntax tree and control flow graphs. Our CC calculation approach is expressed by equation (1) to (3),where lowercase cc represents the cyclomatic complexity at the function or file level.The uppercase CC represents the overall cyclomatic complexity of the software package,serving as a crucial factor in our tool.Here, |E| denotes the number of edges in the function control flow graph,and |N| represents the number of nodes.The function control flow graph is constructed based on branching structures such as if, while, and switch statements.

$$cc(function) = |E| - |N| + 2 \tag{1}$$

$$cc(file) = \sum_{i=1}^{n} cc(function) \tag{2}$$

$$CC = \sum_{i=1}^{k} cc(file) \tag{3}$$

3.1.2 Arch_code scan function. Other tools only consider x86 architecture to locate architecture-related code for statistical analysis. We propose a new idea that explore whether multiple architectures can be selected or if other architectures can be individually targeted. Based on this, we conducted comparative experiments using various architectures, including X86, ARM, MIPS, RISCV, SPARC, and PowerPC. We performed scanning and counting based on the keyword and function dictionaries specific to each architecture, which were derived from common patterns summarized by official websites. By analyzing the correlation coefficient (Pearson's) between the keyword coverage line statistics of the six architectures and the score representing the porting complexity, we found that the correlation ranged from 0.44 to 0.58, indicating all correlation results are not significant. Additionally, the correlation between the total values for all architectures and score was only 0.49. Based on these results, analyze that our tool try to effectively assess the porting difficulty between two architectures. It is not as simple as adding up the modification amounts for different architectures, as it may only reflect an increase in code volume. Moreover, due to the large differences in row-based statistical results caused by the differences in code structure reflected by each architecture, this approach that easily considers all architectures simultaneously without considering the code structure and architectural weight lead to ambiguous outcomes.Therefore, based on these experiments, we still select x86 to build Arch_code scan function, given its mainstream position.

We form the Arch_code dictionary: (1)*Conditional compilation and architecture macros* Use conditional compilation statements, such as "`if defined`", "`ifdef`", in conjunction with all possible architecture macros, "`__x86_64__`". We use an algorithm similar to bracket matching to cover all the code within macros, avoiding the impact of nested layers. (2)*Assembly* 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; (3)*Intrinsic and Builtin* We compile dictionaries of Intrinsic and Builtin functions based on official documentation from x86[6].In the future,we may assign difficulty weights to them. (4)*System calls* 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 and collected currently. (5)*Buildscripts* We locate architecture keywords in the build scripts.

3.2 Evaluation

3.2.1 Dataset. We used the project upstream warehouse to build the data set,which mainly targeting C and C++ are from the OpenEuler 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

Table 2: Experimental results of models

	Accuracy	Pmacro	Rmacro	F1macro
Adaboost	0.786	0.594	0.775	0.650
Random Forest	0.857	0.817	0.873	0.836

Table 3: Tool effectiveness comparison from community feedback

Porting Complexity	Du's Tool		RAX	
	TP	FN	TP	FN
Low	32/37	5/37	35/37	2/37
Medium	5/8	3/8	6/8	2/8
High	2/6	4/6	6/6	0/6
Total	39/51	12/51	47/51	4/51

can be found in 2.1. The evaluation scores are divided into three categories: low, medium, and high[5].This includes 139 projects as training, testing, and validation datasets, while more than 70 additional projects were used as a tool effectiveness evaluation dataset. To gather insights from developers, the community, forums, and emails were utilized, resulting in over 51 meaningful feedback responses. In order to address the issue of data imbalance caused by collecting datasets solely from the openEuler mailing list, where more than 80% of the projects were initially evaluated as having low porting complexity, a greater emphasis was placed on projects that had higher numbers of likes on GitHub and were deemed more important and active. Additionally, efforts were made to collect projects from the system bottom such as operating systems, kernel-level projects,mathematical computation libraries,and machine learning libraries to address RQ 2.

3.2.2 Result and Analysis. To answer RQ 2, based on the work of Chen Xiang et al. [14], aiming at the status quo of unbalanced datasets and small sample datasets, we try to use ensemble learning methods [25], SMOTE [20], 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 model as the classifier, as shown in Table 2.

We continued to expand 72 projects as a supplementary data set, and the prediction results were determined through comments from the community and forum. Github presents comparison[4] with the porting complexity evaluation tool developed by Du et al. across various projects, the comparison results are shown in Table 3.Through real-world evaluations within the developer community, our tool has exhibited a significant improvement in accuracy, surpassing other comparable tools by a margin of 15.6%. In Github, we provide a comprehensive compilation of the community's valuable feedback comments, accompanied by the relevant web links for further reference and validation.

After adding the code complexity factor and modifying the original model, RAX's result prediction accuracy is improved. This is because code complexity can clearly distinguish three types of 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. Another reason is that we have added more medium or difficult complex samples to the data set to increase robustness and improve the innovativeness of the tool.

Within evaluation, there are instances of inaccurate predictions. Analysis of some variations reveals that certain projects exhibit inherent complexity, entailing a substantial presence of architecture-specific code defined by Arch_code. Nevertheless, the practical porting process benefits from an array of auxiliary tools designed to support developers. In response to valuable suggestions from developers within the OpenEuler community, we conducted an evaluation of the porting complexity for various versions of the Chromium.Initially, we were able to accurately assess the porting complexity, yielding promising results. However, when it came to evaluating the work involved in iterative updates between versions, existing tools proved insufficient for precise estimation.Sometimes developers design their work with generic code for unknown architecture and CPU architecture parts are well separated from the common code,that just need only to adjust the moderate amount of code to recognize RISC-V,like nuttx.

4 RELATED WORK

With the development of the RISC-V ecosystem, projects of porting for RISC-V has attracted much attention in both academia and industry. For example,Tine Blaise et al.[24] accomplished the porting of the OpenCL framework to commodity RISC-V, thereby expanding RISC-V's accessibility to a wider range of scientific computing applications. Cao Hao et al.[21]designed an automated porting framework to solve the problem of efficient porting of basic mathematics libraries. However,there is still a lack of fine-grained assistance tools for architecture portability evaluation. Alvares et al. [10] proposed using the development team's "tolerance" for code complexity to infer team development capabilities and performance indicators for projects. Vard et al. [7] emphasized that the internal quality of software impacts developers' capabilities and maintenance duration. They conducted a comprehensive study on various existing code complexity metrics' validity and suggested incorporating empirically observed code features as complexity triggers.We have pioneered the application of code complexity metrics in the domain of porting assessment, while existing research primarily focuses on code refactoring and defect detection[9].

5 CONCLUSION

The software porting work for RISC-V is being actively promoted, but it still faces the challenge of over-reliance on expert knowledge.We propose RAX, with the aim of recommending projects of varying porting difficulties to developers. Our method incorporate the code complexity and architectural code to reflect developers' actual workload. We verified the effectiveness of the evaluation

of 51 extension projects in real porting scenarios through their respective communities or forums. Observed significant performance improvements compared to existing tools.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 original tool, designing more granular statistical methods for architecture-related factors.

REFERENCES

[1] [n.d.]. OpenCloudOS adds support for RISC-V. ([n. d.]). https://baijiahao.baidu.com/s?id=1742039074702920281&wfr=spider&for=pc

[2] [n.d.]. RISC-V has officially become openEuler's official support architecture. ([n. d.]). https://forum.openeuler.org/t/topic/3110

[3] 2023. openEuler:riscv64:BaseOS:stage2. (2023). https://build.openeuler.openatom.cn/project/show/openEuler:C

[4] 2023. porting complexity evaluation details. (2023). https://github.com/wangyuliu/RAX-2024/Data

[5] 2023. porting complexity experimental evaluation. (2023). https://github.com/wangyuliu/RAX-2024

[6] 2023. x86 Intrinsic. (2023). https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[7] 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).

[8] 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*.

[9] 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.

[10] 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

[11] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abughararh, 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

[12] 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*.

[13] 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.

[14] 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

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

[16] Du Jiman. 2023. Design and implementation of a RISC-V porting task planning system based on RPM source package dependency. (2023). https://gitee.com/randomwebsite/risc-v-porting-task-planning-system

[17] He Lei. 2016. Code Complexity Based Software Evolution Evaluation and Analysis. (2016).

[18] 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.

[19] 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

[20] 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

[21] 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)*.

[22] Sholiq, R A Auda, A P Subriadi, A Tjahyanto, and A D Wulandari. 2021. Measuring software quality with usability, efficiency, and portability characteristics. *IOP Conference Series: Earth and Environmental Science* 704, 1 (mar 2021), 012039. https://doi.org/10.1088/1755-1315/704/1/012039

[23] Touseef Tahir, Ghulam Rasool, and Cigdem Gencel. 2016. A systematic literature review on software measurement programs. *Information and Software Technology* 73 (2016), 101–121. https://doi.org/10.1016/j.infsof.2016.01.014

[24] 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

[25] 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

[26] 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).