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://youtu.be/g_e4VhG4kkM.

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[25].Many distribution manufacturers are actively promoting the porting of the RISC-V, for example,OpenEuler has joined RISC-V International organization, while the openEuler 23.09 main repository has currently completed the porting work of 5,694 software packages for RISCV64[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[7].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.Even the developer is very familiar with the architectural code, it is impossible to accurately assess the overall workload.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][22][21].However,code complexity metrics such as

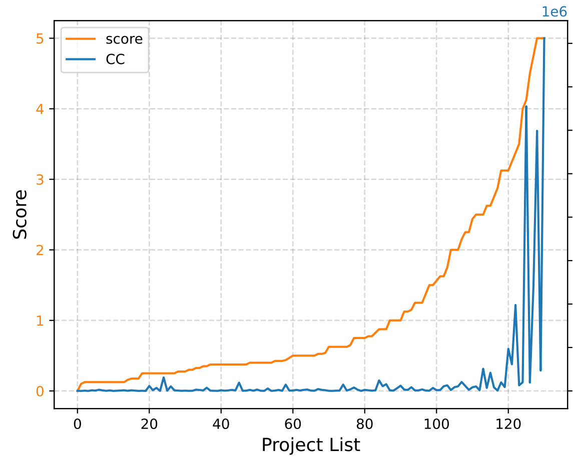

Figure 1: The Relationship between Score and CC

Cyclomatic Complexity are important measurements for software development and maintenance.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 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.

The remainder of this paper is organized as follows.In Section 2,we briefly analyze architecture-related factors. Section 3 presents the implementation and evaluation of RAX. Section 4 and 5 presents related work and conclusions.

2 MOTIVATION

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

RQ 1: How can we select architecture-relevant factors that is fine-grained to design an automated evaluation tool of porting complexity?

RQ 2: How to implement fine-grained and automated tools to help developers accurately assess the complexity of their projects in real world?

As we all known, the metrics of projects porting complexity is influenced by many factors, such as the scale of projects,the logical structure of the architectural code. To answer RQ 1, we introduce the code Cyclomatic Complexity(CC) and Arch_code.

2.1 Factor 1: Cyclomatic Complex(CC)

First,we try to use code complexity metrics to evaluate porting work. CC can help developers make macro judgments on software complexity and maintenance difficulty,and helps the tool improve accuracy [12]. During the porting process, developers will target a large number of project source 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. Based on the above, we use the lizard CC detection tool and optimize its functions.This paper proposes a new CC 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. 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.

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

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

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

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.

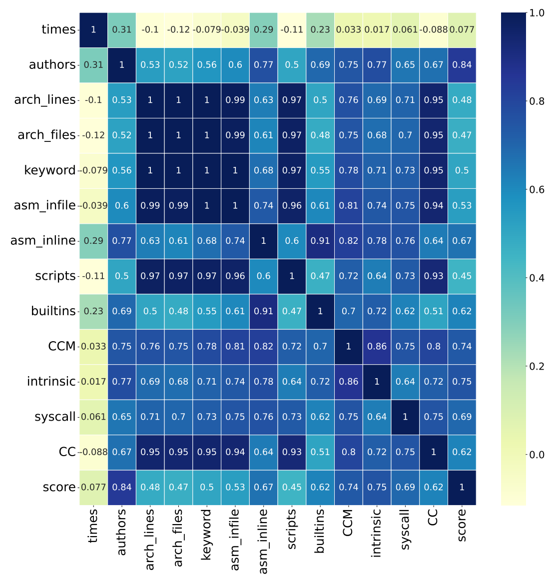

Figure 2: The Heat Map of Pearson Correlation

2.2 Factor 2: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 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; (3)*Intrinsic and Builtin* We compile dictionaries of Intrinsic and Builtin functions based on official documentation from x86[5].In the future,we may assign difficulty weights to them. (4)*System calls* 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.

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

Table 1: Factors Among Different Tools

	CC	Full Assembly	Inline Assembly	Intrinsic	Builtin	System Call	CCM	Buildscripts
Tool[16]	✗	✔	✔	✔	✔	✔	✔	✔
RAX	✔	✔	✔	✔	✔	✔	✔	✔

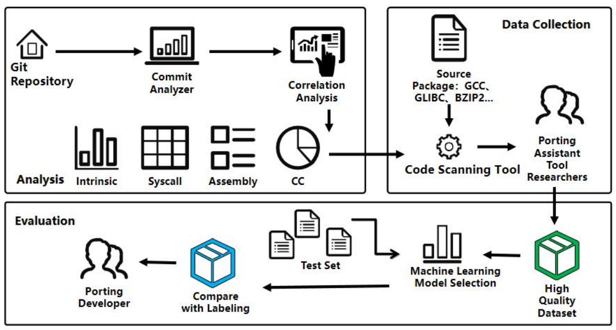

Figure 3: Flow Chart of RAX

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.

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

3 RAX

Since we identify architecture-relevant important factors affecting porting efficiency, we implement a fine-grained and automated tool to evaluate complexity. Figure 3 shows the entire technology roadmap in three parts such as Analysis, Data Collection and Evaluation. RAX can be deployed easily by installing dependencies, highlighting its convenience and user-friendliness.

3.1 Implementation

3.1.1 Analysis. We intend to use the Lizard tool, which employs a methodology consistent with the McCabe's method[19], to obtain CC metrics. However, Lizard tool only 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 [10].We need to redefine the CC method to avoid the use of complex methods, which will cause deviations to affect the classification results.Additionally, we must address the potential performance overhead problem caused by converting the entire project into abstract syntax tree.

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.

3.1.2 Dataset Collection. We used the project upstream warehouse to build the data set,which mainly targeting C and C++ are from the OpenEuler list and GitHub[3].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. It is worth emphasizing that, all these projects' complexity evaluated by RAX hoped to be verified by the own developer of projects.

3.1.3 Evaluation. In our study, we opted to utilize Random Forest machine learning model for conducting training and testing to obtain the final classifier model. The extended dataset was used as an additional input for the classifier to acquire evaluation results. We subsequently performed evaluations on 72 projects and sought community feedback during the tool's effectiveness evaluation phase.

Table 2: Results of Different Evaluation Tool

	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 Based on 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

3.2 Evaluation

3.2.1 Dataset. To evaluate RAX, We chose typical and popular C and C++ projects from upstream warehouse of the OpenEuler and GitHub[3]. First, we searched keywords "rv" or "riscv" and collected 139 projects as our training and testing dataset. Then, five postgraduate students are invited to tag the porting complexity for these projects. The evaluation scores are divided into three categories:low, medium, and high[4]. Second, models are trained and tested based on the 139 projects. Third, to evaluate the RAX, additional 72 projects are collected and evaluated from GitHub. To gather response from developers, the community, forums, and emails were utilized.

3.2.2 Result and Analysis. To solve the problem of unbalanced datasets and small sample datasets, we used ensemble learning methods [24], SMOTE [20], and random oversampling methods for data preprocessing[14] addressing.By comparing multiple indicators, we select the Random Forest, which had the highest accuracy of 0.857 as the classifier, as shown in Table 2.

We provide a comprehensive compilation of the community's valuable feedback comments, accompanied by the relevant web links for further reference and validation shown in [4] and Table 3. Our experiments shows that, compared with other tool, RAX is more accurately to evaluate projects' complexity. In detail, the true positive (TP) of other tool is 39/51, while the RAX is 47/51. At the same time, the RAX has a lower rate of false positive (FN) than other tools. The values are 4/51 and 12/51, respectively. 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%.

We conducted an in-depth analysis of the essential reasons for the improvement in accuracy after adding CC factors.This is because code complexity can clearly distinguish three types of porting difficulty.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. (1)In cgal project, developers believe that some functions or performance improvements are only worth the effort in scenarios where users need them, and cgal can be made to work properly with a small amount of modifications.Since our tool traverses the global code,and optional information cannot be identified.(2)In PortAudio project, the improved portability is attributed to the existence of equivalent C language implementations for the assembly code. This has a significant impact on our evaluation results because RAX can not recognize the semantic meaning of the code.(3)Developers believe that dlib project has implemented distribution versions of multiple architectures, and the porting experience reduces the difficulty of porting.(4)Sometimes CPU architecture parts in projects are well separated from the common code,that just need only to adjust the moderate amount of code to recognize RISC-V,like nuttx project.

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.[23] 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.[11]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. [9] proposed using the development team's "tolerance" for code complexity to infer team development capabilities and performance indicators for projects. Vard et al. [6] 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[8][13].

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

[6] 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).

[7] 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*.

[8] 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.

[9] 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

[10] 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

[11] LIU Dan XU Jin-chen CAO Hao, GUO Shao-zhong. 2021. Automatic Porting of Basic Mathematics Library for 64-bit RISC-V.

[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] 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

[22] 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

[23] 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

[24] 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

[25] 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).