Migrating Software from x86 to ARM Architecture: An Instruction Prediction Approach

Blake W. Ford
bf1109@txstate.edu
Texas State University
San Marcos, Texas

Apan Qasem
apan@txstate.edu
Texas State University
San Marcos, Texas

Jelena Tešić
jtesic@txstate.edu
Texas State University
San Marcos, Texas

Ziliang Zong
ziliang@txstate.edu
Texas State University
San Marcos, Texas

2021 IEEE International Conference on Networking, Architecture and Storage (NAS) | 978-1-7281-7744-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/NAS51552.2021.9605443

***Abstract*—For decades, the x86 architecture supported by Intel and AMD has been the dominate target for software development. Recently, ARM has solidified itself as a highly competitive and promising CPU architecture by exhibiting both high performance and low power consumption simultaneously. In the foreseeable future, a copious amount of software will be fully migrated to the ARM architecture or support both x86 and ARM simultaneously. Nevertheless, software ports from x86 to ARM are not trivial for a number of reasons. First, it is time consuming to write code that resolves all compatibility issues for a new architecture. Second, specific hardware (e.g. ARM chips) and supporting toolkits (e.g. libraries and compilers) may not be readily available for developers, which will delay the porting process. Third, it is hard to predict the performance of software before testing it on production chips. In this paper, we strive to tackle these challenges by proposing an instruction prediction method that can automatically generate AARCH64 code from existing x86-64 executables. Although the generated code might not be directly executable, it provides a cheap and efficient solution for developers to estimate certain runtime metrics before actually building, deploying and testing code on an ARM-based CPU. Our experimental results show that AARCH64 instructions derived using prediction can achieve a high Bilingual Evaluation Understudy (BLEU) Score. This indicates a quality match between generated executables and natively ported AARCH64 software.**

***Index Terms*—x86 Architecture, ARM Architecture, Software Portability, Instruction Prediction, Language Translation**

I. Introduction

Over the past few decades, x86 has undoubtedly become the dominate architecture for software running on modern CPUs, thanks to the long term support from both Intel and AMD. This trend is shifting quickly with ARM emerging as a highly competitive CPU architecture. Recently, Apple has made a significant investment in ARM by designing its own ARM-based System-on-a-Chip (SoC) hardware. Announced in late 2020, the M1 Chip will be used to power all its primary products (e.g. MacBook, iMac, and iPad) [1]. Less than a year later, Microsoft also announced its plan to further endorse the ARM architecture in Windows 10 and develop comparable hardware [2]. Considering the enormous size of Apple and Microsoft's software ecosystem, it is inevitable that significant amounts of software will be fully ported to ARM or support both x86 and ARM simultaneously.

However, migrating software from x86 to ARM is not a trivial task. Currently, there are two primary strategies available to developers. The first strategy requires developers to natively port and create some new, dedicated code to support the ARM architecture. The learning curve to fix all compatibility issues could be high and extremely time consuming for inexperienced programmers. Alternatively, developers can leverage binary translation and run their code under emulation (e.g. Apple Rosetta [3]). Although this approach introduces minimal development cost, it also overwhelmingly favors compatibility over performance. Software performance could degrade by 20% or more due to the overhead incurred through emulation [4]. Completely relying on emulation or binary translation is also risky because there is no guarantee as to how long the underlying tech will be available. For example, Apple has previously set a precedent for discontinuing these surrogate forms of execution in subsequent OS releases [5]. For either strategy, the porting process could be further delayed if the hardware (e.g. ARM chips) or supporting toolkits (e.g. libraries and compilers) are not provided in a timely fashion. To the point, obtainable test hardware was not available until six months after Apple announced the M1 chip. Without access to this hardware, it is almost impossible for developers to predict the performance of their code on the AARCH64 architecture.

To address weaknesses of current methods for porting software from x86-64 to the AARCH64 architecture, we propose an instruction prediction method, which can automatically generate an ARM version of existing x86-based software. Although this code may not be directly executable, it does not require software developers to obtain physical hardware or have in-depth knowledge of cross-platform development. Therefore, it provides a cheap and efficient solution for developers to estimate certain runtime metrics of their software before building, deploying and testing on an ARM-based CPU.

Figure 1 summarizes the key steps of our methodology. First, a code corpus will be created and shared with all developers. This is essentially a large database that consists of 1000+ source functions/routines built for the x86-64 and AARCH64 architectures respectively. It is used to learn sufficient historical patterns of code and to map relationships between each architecture's instructions. Next, an n-gram model is utilized to create an accurate native translation

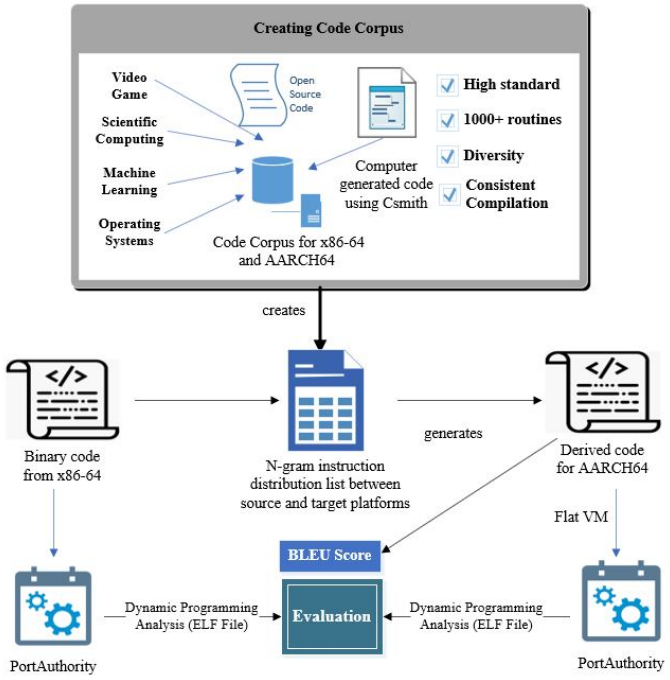

Fig. 1. Overview of the instruction prediction approach.

of the program from x86-64 to AARCH64 according to a learned n-gram instruction distribution list. Last but not the least, we develop the PortAuthority tool that uses dynamic programming analysis to predict the performance of derived code on ARM chips. We comprehensively evaluate our method using the Bilingual Evaluation Understudy (BLEU) Score [6]. The experimental results show that derived code using our method can achieve a high BLEU score, which provides strong evidence that our method can generate AARCH64 instructions from x86-based software with quality near a native port.

II. Related Work

It is unrealistic to judge software performance without a due diligence porting trial and some upfront, hand-coded work. Predicting the runtime behavior of software conventionally requires executing some sample code on a device of interest. When estimating cross-platform software performance, a developer needs to be confident the source can be cross-compiled and that the resulting code will run deterministically. This section reviews the strengths of portable code and introduces some related work on executable prediction.

A. Portability

Software exhibits portability when its adaption costs are less than those of redevelopment [7]. Rather than by design, there is the common but poor practice of applying ad hoc methods when the eventual need for a port is discovered. Economically, this works against sound investment principles in a product line destined to remain in the market for a prolonged period.

Several obstacles prevent software from being easily ported to a new device with different architecture. For example, if a code base relies on language features not available in the compiler for a required target, significant rework could be required to enable related functionality. Newer standards are a frequent source of non-compliance in a language or standard library, but other historical sources may also exist [8]. Software issues can also exist beyond the build process and continue into the runtime environment. System RAM on some targets may be non-expandable and designed to support a finite number of simultaneous applications. In contrast, desktop computer components tend to be expandable because the expected experience on that device includes the possibility of an arbitrarily large, multitasking workflow. Because most compilers are not designed to consider variable device specification, when software is ported a developer might be surprised by incompatibility.

B. Cross-platform Prediction

Broadly methods for cross-platform prediction are not new [9] [10] [11]. Methods we reviewed for this paper favor examining code from the bottom up or at near the binary level. This allows for insights to be gained from code without the benefit of source. It excludes the possibility of reporting false information to a user that may not be knowable early, apparent only after later compilation or linking stages of the build pipeline. In emerging platforms, mature tooling and ample code may not be readily available to address prediction with supervised learning [12]. Conversely, there are techniques with roots in traditional language processing that do not require big data. Asm2vec is an example of a cross-platform utility designed to find a user's code within random executables [10]. Based off the widely popular Word2Vec algorithm, this tool uses a neural network approach to process an entire application searching for sections of code with bilingual equivalency to a given input. Contiguous sequences of n items, or n-grams, have been used successfully in this family of pattern matching techniques with binaries compiled using similar levels of optimization [11].

While we acknowledge other methods for cross-platform prediction, many are not directly comparable to our method because they are based on vectorization not generation. Moreover, most searching methods require complete binary data from both the source platform and the target platform, which is hard (if possible) to obtain. The closest work to our proposed method was published by Tumeo [9], which was capable of providing software metrics early in the development process without access to specific hardware.

C. BLEU Score

The Bilingual Evaluation Understudy (BLEU for short) score [6] is a widely accepted metric for evaluating the quality of a generated sentence to a reference sentence. It is always in the range of 0 to 1, with 1 being a perfect match (generally unobtainable) and 0 being a perfect mismatch. Although the BLEU score is mostly used for natural language translation, we use it to evaluate the quality of code translation from an x86-64 source to an AARCH64 target.The generated code for

the target platform is known as the candidate. When comparing a candidate to its reference (a native AARCH64 port), we window over the candidate combining terms at a distance of *n*. We look for instances of that n-gram in the reference code where results are based on statistical precision.

III. Methodology

To reduce the burden of developers for migrating software from x86 to ARM, we propose a novel technique that can automatically generate ARM-based code from x86-based code and estimate certain runtime metrics of derived ARM code without building, deploying and testing them on an AARCH64 chip. In this section, we provide a detailed explanation about the key components of our method, which includes an n-gram instruction prediction model, statistical binary occurrence data, the PortAuthority tool, and the Flat VM.

A. N-gram Instruction Prediction Model

The primary responsibility of the n-gram instruction prediction model is to predict what instructions will be the most probable compiled for AARCH64 based on the original instructions used on the x86-64 source platform. To increase the accuracy of calculating such probability, it is necessary to have a large enough code corpus (i.e. a large database that consists of many identical functions built for each target architecture) to reveal sufficient historical patterns of code mapping relations between x86-64 and AARCH64.

In our method, we create the code corpus by following the high data standard presented in [13]. More specifically, we assembled a set of C/C++ routines compiled using a specific compiler version common to both x86-64 and AARCH64. It is critical that these executable segments are created using the same process per platform and using identical source code across platforms. In addition to available open source code, we also incorporate a selection of Clang compiler tests and computer generated C code using Csmith [14]. Each function in our code corpus is compiled using the Clang compiler from the LLVM project at commit hash *bb7a57*.

When the code corpus is completed for both x86-64 and AARCH64, a dictionary for cross-platform translation becomes available. The BLEU score used to evaluate our model is normally applied to written text. The score is typically calculated from each "sentence" and averaged over the entire corpus of input. We have scored our translations likewise where each function translated is treated as a "sentence". All "sentence" scores are then averaged and presented as our final results.

However, it is worth noting that limiting software diversity in the code corpus or mixing compilation processes will decrease the accuracy of our prediction model. Using the compiler(s) intended to build a cross-platform project is also pivotal. The method will work using two different compilers provided that one compilation process is exclusively used to create the corresponding piece of the database on each of the platforms.

Drawing from the code corpus, we can window across *n* instructions at a time recording the x86-64's n-gram and the corresponding ARM architecture positions. We build a special tool to process the entire dataset and convert the final findings into a map (see Table 1 as an example). This output is composed of a list of AARCH64 n-gram frequencies attached to each discovered x86-64 n-gram from the code corpus. Using these distribution lists, we emit the respective cross-platform version according to the given probability.

TABLE I
SOME AARCH64 2-GRAM DISTRIBUTIONS FOR X86-64 PUSH-MOV

Bigram	Frequency %
SUB-STP	45
SUB-STR	16
STP-STR	6
MOV-STP	5
STP-MOV	5

B. Statistical Occurrence Data

One important issue that must be addressed when translating executable code is the potential difference in instruction density between the source platform and the target platform. For instance, the number of AARCH64 instructions in a given program is typically larger than that of it's x86-64 counterpart. In our code corpus, we have observed that on average AARCH64 variants of each program contain around 5% more individual instructions than those built for x86-64. Therefore, generating an executable with a 1:1 ratio of instructions often ignores a certain percentage of instructions in the original program. To solve this problem, we supplement our raw n-gram sequences with statistical occurrence data. Each generated executable is padded after the initial n-gram conversion with an additional short list of raw probabilistic instructions equal to the deficit expected between AARCH64 and x86-64.

C. PortAuthority

To derive metrics information from an executable's instructions without direct measurement, we include a dynamic program analysis tool called PortAuthority [15]. First a user needs to build detailed execution information for their application through one of a few frontends. The purpose of a frontend is to single step and record each instruction a user's program executes for a given range. After a profile is recorded, users can then load their data into the hosted PortAuthority application to visualize insights related to their code.

D. Flat VM

While PortAuthority can directly process runnable ELF format executables built by any compiler, it lacks the ability to predict program behavior based on a less than complete representation of a program. To mitigate this problem, we introduce Flat VM that allows us to work around PortAuthority's runnable input process limitation. Specifically, we inject raw binary data into incomplete ELF files using the *objcopy*

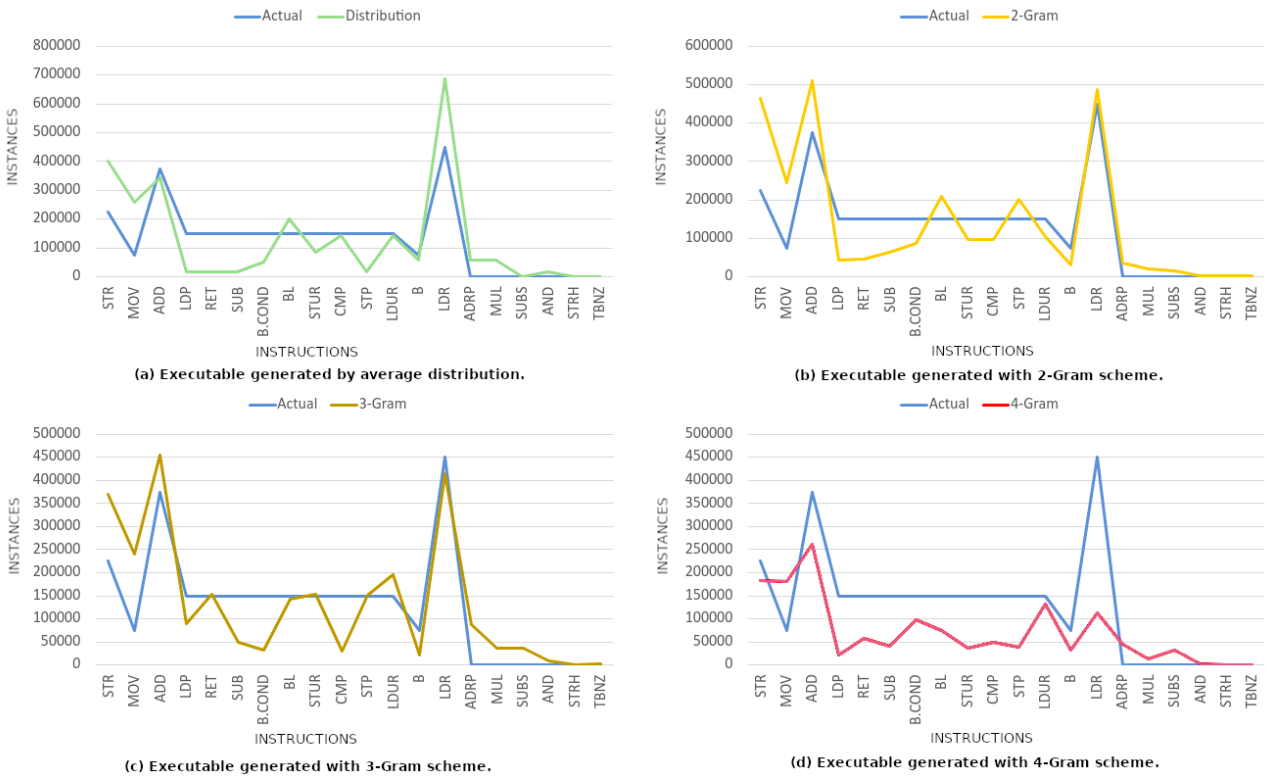

Fig. 2. Generated instruction instances using different methods versus instruction instances from the full port of the Fibonacci(24) test.

utility with the *update-section* option. This tactic allows us to override the text section from a basic C program with our generated instruction data. While not executable on its own due to limitations patching faults caused by ABI non-compliance, branch endpoint changes and stack corruption the resulting ELF can be read and processed by off-the-shelf tools, like the GNU Binutils. To tap into the profiling capability provided by PortAuthority without the benefit of single step execution, we create a simple virtual machine to process our ELF executable into the intermediate format created by the other profiler frontends.

The JSON input format is divided into two distinct parts, the fixed header and variable content. The header is small, containing only a few values related to platform identification and size. The bulk of the input data is composed of the content, per instruction objects with three member variables; *a*, *o*, and *m*. In our Flat VM program, for the length of the text section, we output this structure and then increment a program counter by the size of the last instruction processed.

IV. Experimental Results

In this section, we conduct a series of experiments to comprehensively evaluate the effectiveness of our n-gram generated executables using a variety of standards and over multiple values of *n*. A key point to understanding our results is awareness that we have no direct control over how the ported application is formed in this process. The blue lines shown in Figure 2 and Figure 3 represent an externally compiled reference binary that would not be present in any final instruction prediction based tools. First we look at the differences between the number and categories of generated instructions versus actual instruction mix in a full application port. Next we consider the predicted instructions matching the exact placement within a fully ported executable. Finally we calculate the BLEU score of these binaries to determine whether or not they would be considered good translations. Four test applications are run through the same series of tests and a summary of the results are shown in Table 2.

A. Context Free Tests

Our first test program is a short Fibonacci sequence calculator. With all our examples, we try to sample an area of 2-4 million instructions in order to keep consistency across our results. While the execution of each is highly detailed, the expected runtime of any examples is only a few seconds. Outcomes from analyzing this program are shown in Figure 2, which illustrates the worse case differences in instruction instances between our generated ELF files and a completed AARCH64 port of the Fibonacci calculator. The absolute value of instructions in the completed port versus the instances provided by our model are charted. Gaps are shown in decreasing order left to right and limited to the worst 20 unique instructions within the program. The executables in our test catalog are composed from 30 to 60 unique instructions per application.

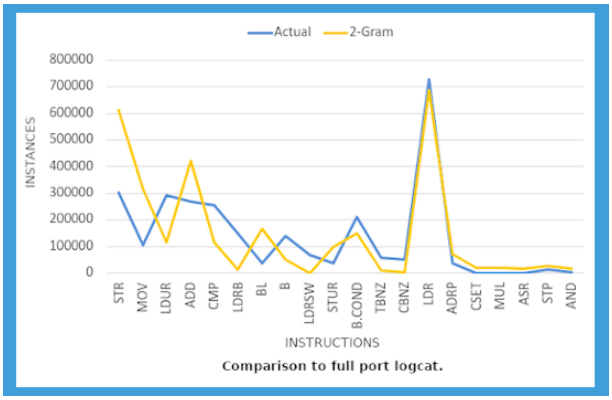

Fig. 3. Generated 2-gram instruction instances versus instruction instances from the full port of logcat.

In the first experiment, the generated executable is composed only from the likely statistical distribution of instructions available from our instruction database. We consider inputs like program size and relative architecture instruction density but no code from an original non-architecture program is referenced. The results are shown in Figure 2a. This result acts as a control on which we built further evaluations. Note how even without a compiled executable for reference, the overall structure of the program is loosely in line with a true port. This emphasizes that repetitive elements involved in developing a program may overshadow custom computation internally, especially when considering all of these programs conform to the same requirements for calling functions, application binary interfaces and other forms of compliance code.

Moving on to the next variant, Figure 2b provides comparative results for the 2-gram version of the same generated application. Unlike the first generation scheme, based purely on statistical distribution, all latter schemes use an x86-64 version of the core application as a source for emitting equivalent n-grams. The scale of differences between the candidate and the reference instruction instances drops sharply where n-grams are used. Immediately, the outline of the 2-gram form snaps tighter to the instruction mix exhibited by the full port. Unfortunately, this momentum is not clearly maintained as we increase values for n, as evidenced by the diminishing return of the 4-gram prediction. The trend of worst case instruction improvements accompanied by loss of cohesion on instances originally further to the right on our chart holds true as n increases. This amplified noise is visible in Figure 2c and Figure 2d.

The experiments are conducted on the remaining 3 test programs with similar results. Our second test program is based on code from the logcat application included with the Android operating system. Logcat provides a mechanism for filtering and prioritizing messages familiar to users of system logs. We pull from commit hash *cfaded* and make changes to this code in order to pump messages directly so that the program could be tested without the need for a second application for input. Next we include a routine from the TensorFlow framework, an open source platform for machine learning. Our test uses code for single-threaded matrix multiplication from version 1.4.1 with some infrastructure to support execution outside of the full framework. Finally we test using a game project designed for the Arduboy handheld game console. Originally Sirene was written for an 8-bit embedded CPU architecture but here is compiled on x86-64.

Closely matching raw instruction mix is valuable when profiling code through context unaware tools like PortAuthority. Other binary analysis tools may benefit from accurate context, but this tool provides estimates based on per instruction signatures. For analysis without context, we would suggest equally 2 or 3-gram interpretations as input from our tools. Some of our test applications perform slightly better with the 3-gram interpretation but the results are close. While exact matches for instructions are preferred, PortAuthority has proven that categories of instructions have similar signatures, therefore some misalignment is easily tolerated. In practice, having a number close to the overall instructions and with proper categorization will yield usable results. Related 2-gram information for the best test program is shown in Figure 3, which follows the same design and structure as discussed in Figure 2.

TABLE II
RESULTS USING 2-GRAM GENERATION FROM OUR TEST CATALOG

Test	Exact Match %	BLEU
Fibonacci (24)	8	0.22
logcat	10	0.79
TensorFlow (MatMul)	10	0.33
Sirene	6	0.42

B. Context Aware Tests

While PortAuthority does not currently require context to work as designed, other forms of binary analysis do. Context is important as it may be required to determine software metrics. Cache utilization is an example of context aware analysis possible through instruction profiling [16]. We conduct experiments to study the impact of n-gram generation on context preservation (ref. Table 2 for details on exact matches and each generated program's BLEU score). Specifically, we recycle the same test programs for use in our context quality tests. Here we focus on the preservation of execution context and evaluate in both a strict and a lenient mode. In strict context evaluation, we analyze the accuracy of generated instructions in relation to addresses assigned within the reference port. We only consider exact matches for this value. Lenient evaluation is based on the BLEU score, which indicates computational readability. Acceptable BLEU scores relate to fluency thus in this domain should also relate to the preservation of execution context. Expectations for readability can be viewed as appreciably better for each 10% improvement in the score. Scores that are less than 0.20 usually do not provide significant value while scores above 0.60 offer the highest fluency [17]. The score penalizes words that appear in the candidate more times

than its references. Hence there is a high level of correlation between the closest modeling executable tested (see Figure 3), and its BLEU score (ref. Table 2).

```
if (pressed)
401042:       f6 45 fd 01             testb  $0x1,-0x3(%rbp)
401046:       0f 84 0d 00 00 00       je     401059
{
  arduboy.audio.saveOnOff();
40104c:       e8 2f 00 00 00          callq  401080
  gameState = STATE_MENU_MAIN;
401051:       c6 04 25 33 e1 60 00    movb   $0x1,0x60e133
401058:       01
}
```

Fig. 4. Mixed disassembly for Sirene.

C. Discussion

One limitation to improving our BLEU scores is the lack of several reference translations to compare against as compilers should always build source in the exact same way. In contrast, a human translator would not be held to the same standard.

We recommend 2-gram instruction prediction when profiling ELF executables as it generates the best result and fits for broad categories of analysis. Exact matches are never greater than 10% in any of our tests. A minimum BLEU score of 0.20 is maintained over all experiments, which suggests the gist of a given program is clear, albeit with significant ordering errors. This compliments the results presented during our context free testing. Most of our translated executables provide good, understandable translations and one test even achieves the high quality standard.

Apineni et al. reported that natural language translations can achieve their highest levels of n-gram correlation around n values of 4 [18]. Functional blocks may contribute to our comparatively low best value for n of 2. Consider the mixed disassembly sample in Figure 4. The *objdump* utility allows user to create this output laying out certain compiler decisions used to create the program. Notice that in each of these 3 lines of code only 1 or 2 instructions are required to convey the source intent. If the majority of source functionality is built using only a couple of instructions, it makes sense that larger n-grams would fail to produce consistent results. Ultimately, trusted values are more important than high numbers for n. Where possible, clarity may be improved by translating from the platform with the lowest instruction density.

ELF files can be generated for a variety of platforms. Here we work with what we feel to be the two most contemporary choices for cross-platform development, x86-64 and AARCH64. However, there is no known limitation to working with upcoming platforms like RISC-V. Conducted properly, we expect the method to function well with all ELF generating targets.

V. Conclusion and Future Work

As ARM is quickly emerging as a highly promising architecure to compete with x86, it can be expected that a copious amount of software will be migrated to the ARM architecture in the near future. To facilitate the process of migrating software from x86-64 to AARCH64, we propose an n-gram based instruction prediction approach to estimate runtime metrics without building, deploying and testing code on ARM chips. Our experimental results show that the 2-gram instruction prediction can provide quality portable code with desirable BLEU scores, a measure of readability. We have shown the potential for use with context-free analysis as well as some exciting applications in the context-aware space.

References

[1] F. Lardinois, "Apple announces the m1, the first chip in its apple silicon family," 2020. [Online]. Available: https://techcrunch.com/2020/11/10/apple-announces-the-m1-the-first-member-of-its-apple-silicon-family/

[2] A. Bacchus, "Microsoft goes all-in on windows 10 on arm," 2021. [Online]. Available: https://www.digitaltrends.com/computing/microsoft-goes-all-in-on-arm-build-2021/

[3] A. Inc., "About the rosetta translation environment," 2020. [Online]. Available: https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment

[4] F. McShan, "Performance of rosetta 2 on apple m1," 2021. [Online]. Available: https://mjtsai.com/blog/2020/11/16/performance-of-rosetta-2-on-apple-m1/

[5] "Inside mac os x 10.7 lion: Missing front row, rosetta and java runtime," 2011.

[6] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[7] J. D. Mooney, "Bringing portability to the software process," 2000.

[8] Corob-Msft, "Microsoft c++ language conformance table," 2019. [Online]. Available: https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-language-conformance?view=vs-2019

[9] A. Tumeo, "Architecture independent integrated early performance and energy estimation," in *IGSC 2017*. IEEE, Oct 2017, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8323602

[10] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[11] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn," *Applied Sciences*, vol. 9, no. 19, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/19/4086

[12] I. Moura, G. Pinto, F. Ebert, and F. Castor, "Mining energy-aware commits," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 56–67.

[13] M. Post, "A call for clarity in reporting BLEU scores," in *Proceedings of the Third Conference on Machine Translation: Research Papers*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 186–191. [Online]. Available: https://www.aclweb.org/anthology/W18-6319

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[15] B. W. Ford and Z. Zong, "Portauthority: Integrating energy efficiency analysis into cross-platform development cycles via dynamic program analysis," *Sustainable Computing: Informatics and Systems*, p. 100530, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537921000238

[16] I. Games, "Cachesim," 2017. [Online]. Available: https://github.com/InsomniacGames/ig-cachesim

[17] A. Lavie, "Evaluating the output of machine translation systems," 2011. [Online]. Available: https://www.cs.cmu.edu/ alavie/Presentations/MT-Evaluation-MT-Summit-Tutorial-19Sep11.pdf

[18] K. Papineni, S. Roukos, T. Ward, J. Henderson, and F. Reeder, "Corpus-based comprehensive and diagnostic mt evaluation: Initial arabic, chinese, french, and spanish results," in *Proceedings of the Second International Conference on Human Language Technology Research*, ser. HLT '02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, p. 132–137.