AN INSTRUCTION PROFILING BASED FRAMEWORK TO PROMOTE SOFTWARE PORTABILITY

by

Blake W. Ford, M.S.

A dissertation submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
with a Major in Computer Science
May 2022

Committee Members:

Ziliang Zong, Chair

Apan Qasem

Jelena Tešić

Heping Chen

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

The work is dedicated to our children, caught in the middle of Dad's twenty-year goal. May you be the first to accomplish something big and be the very best at even a tiny thing.

ACKNOWLEDGEMENTS

First and foremost, I would like to acknowledge my dissertation committee. Each member has provided valuable insights and help to me during this process. Also in need of recognition is our Program Director Dr. Anne Hee Hiong Ngu. A special thanks is due to my advisor Dr. Ziliang Zong who picked up the project mid-process and pushed me towards the finish. No person is an island, so I must also mention my family. My partner and children have been incredibly supportive of me during this process. This achievement is as much theirs as it is my own.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
RISC	Reduced Instruction Set Computer
ARM	Advanced RISC Machines
ISA	Instruction Set Architecture
NLP	Natural Language Processing
BLEU	Bilingual Evaluation Understudy
CPU	Computer Processing Unit
RAPL	Running Average Power Limit
API	Application Programming Interface
RAM	Random Access Memory
ELF	Executable and Link Format
IDE	Integrated Development Environment
ASIC	Application Specific Integrated Circuit
AI	Artificial Intelligence
ML	Machine Learning

ABSTRACT

In recent history, most software has been built for a single hardware class and developers rarely needed to consider cross-platform development. With emerging paradigms like Edge computing and the introduction of alternative mainstream processing elements, we have reached a landmark for multi-platform development. Moreover, the barriers to successfully introducing a new CPU architecture are also falling, as demonstrated by RISC-V. This can only lead to greater fragmentation in the ecosystem of development platforms and tools that advocate for software portability lag far behind this trend. A notable example at present is the introduction of Apple silicon and Apple's substantial efforts to migrate software from previously supported Intel CPUs to ARM. Unfortunately, in order to access the interesting features of a new platform, developers will be required to rebuild or port their existing software. Porting software can uncover unexpected behavior and produce uncertain results in code that was previously considered stable. A dearth of development tools geared towards porting software means addressing these problems requires significant work and it is difficult for developers to predict the performance and energy consumption of software early on without testing it on the target hardware. Privileged access to prototypes and limited options for simulation before release further complicate the problem.

This dissertation proposes a cost-effective instruction profiling framework that promotes portable software by allowing developers to estimate certain metrics inherent to their software on other platforms without modifying their code and without direct access

to hardware. Specifically, the framework includes three modules: (1) The instruction profiling module analyzes existing code in an efficient way and produces evaluation reports. (2) The estimation module leverages prerecorded metrics to calculate the performance of this code on another platform. (3) The instruction prediction module uses a machine learning approach to automatically generate cross-architecture code for other possible targets. The framework can forecast various metrics including instruction categorization, code coverage, and resource usage simultaneously. Data produced is comparable to existing off-the-shelf benchmarks but available for multiple platforms after a single profiling pass. Experiments herein confirm performance estimates are within externally established tolerances and powerful enough to provide automated assistance to developers considering a software port between disparate platforms.

I. INTRODUCTION

Toolchains and environments have long provided options that help developers to optimize source code for higher performance or lower memory usage. For instance, environments may suggest removing or updating code. Compilers like GCC and Clang have macro-options which bundle together many smaller unique transformations into more user-friendly levels of automatically applied optimization. Other forms, like link time optimization, can be introduced even later in the build process. These options have wide ranging effects on the generated assembly code. Though these enhancements target speed improvements or memory savings broadly, they do little to describe how the code might perform on a different platform. Unfortunately, very few cross-platform forecasting tools are available to developers.

Without proper programmer assistance, problems may not be immediately discoverable given that source code may translate to hardware in a variety of ways. Optimizations or other compiler flags can radically change the source intent so long as the resulting evaluation is the same. For example, dead code elimination allows a compiler to completely remove sections of code it does not expect to be productive during execution. Lacking the ability to make transformations discretely and automatically through a compiler, another option is to provide commentary on a developer's code so that they could be empowered to make some of these changes themselves. Of course, this information would be most useful to a developer at development time and on their personal hardware. As few environments are setup for cross-platform development, projects normally need to explicitly include additional libraries, run an external program or engineer other code to compensate. Instead, the

common software engineering practice today does not account for software ports which could force an expensive future redevelopment cycle. Investigating portability at the beginning of a project and staying mindful of possible ports throughout the entire software development cycle is more desired and cost-effective long term (Mooney, 2004).

One area where cross-platform development tools are handy is in the creation of systems with Edge computing elements. Edge computing refers to the practice of offloading computational effort from strained central areas to areas on a network where existing utilization is low. There are increasing computational needs that make Edge computing a promising alternative to the cloud. Edge computing is a novel and available solution that can alleviate problems caused by distributed applications as data propagation has become the limiting factor in the expansion of cloud computing. The volume of data being produced in the field is growing at a rate far greater than improvements in communication technology (Lin et al., 2019 and Shi et al., 2016). Normal impacts from this kind of expansion range from increased financial costs to lower quality of service. Devices on the Edge are typically less powerful, so software developed with servers in mind may perform worse than expected in this environment. In addition, emerging areas like Edge computing lack host options based on x86, the dominant server CPU architecture. The demands of this segment favor custom processor designs and the market is held by Instruction Set Architectures (ISA) that are historically modifiable unlike x86. Customization requires a product be either licensable as with the ARM architecture or fully open like RISC-V. This highlights another threat to mono-platform development, large scale CPU architecture transitions.

Over the past few decades, x86 has undoubtedly become the dominant 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) (Lardinois, 2020). Less than a year later, Microsoft also announced its plan to further endorse the ARM architecture in Windows 10 and develop comparable hardware (Bacchus, 2021). Considering the size of their combined software ecosystem, it is inevitable that significant amounts of software will be fully ported to ARM or support both x86 and ARM simultaneously.

Consider that each platform, regardless of CPU architecture, provides only limited execution context transferability to another. The relationships between registers, memory and other CPU associated hardware do not map directly between different platforms. Therefore, there exists a set of non-uniform performance metrics for each device: power, bandwidth, compute capability. For instance, servers may require more energy to operate but could also perform CPU tasks faster than Edge devices. Given this diversity, it is difficult for developers to predict the performance of their software. Most opt instead to test explicitly on the target hardware.

Aside from the obvious ability to do empirical testing, extra performance data may be reported from the underlying device itself. Some examples include platform specific counter registers and variables exposed in the Filesystem Hierarchy Standard in Linux. Integrated data sources provide a variety of metrics, so calculations can be made

considering a substantial number of detailed values from the hardware. Counters can work well in the absence of physical metering as specific actions tracked by these registers influence resource consumption (Treibig et al., 2010). However, performance counters were not designed in a portable way and different problems can arise when attempting to relate their values outside narrow specifications or cross-platform. As counters track the dimensions of an entire platform, it may also be impossible to sort out the demands of a single process on a multi-process host without some interference. Extracting and comparing these data from multiple unique platforms requires a clear duplication of work and several bespoke pieces of test equipment. The scarcity of device hardware also plays a role in the ability to effectively port software when developers rely on direct platform testing or inherent platform features.

In addition to hardware, it is also wise to consider language. With the diversity of high-level programming languages, a good cross-platform profiling solution should not be attached to a specific programming language. Developers use different programming languages for different projects, and perhaps even a mix set of programming languages within a single application. Providing an otherwise flexible profiling standard bound to a subset of languages limits the overall value.

This work is an attempt to start improving weaknesses in the software porting process by providing relevant, interactive and automated programmer assistance at design time without the advantage of real hardware. The framework is intended to be platform agnostic. The research uses two of the most common architectures for demonstration purposes. Code originally built for one platform is used to predict the performance of a future port to another. Specific contributions are as follows.

• **Extracting fine-grain performance feedback from non-instrumented, unmodified binaries.** Better, more detailed range of metrics gathered non-intrusively.

• **Accounting for the entire build process.** Describes the true nature of software in ways that cannot be known at the source code level.

• **Allowing language agnostic profiling.** A plausible solution for use with any software package capable of running on a given platform.

• **Reducing dependence on physical measurement.** Software changes can be evaluated without additional test hardware or features.

• **Lowering the cost to estimate software cross-platform.** Highlight the potential for other devices to run software from a single development platform.

• **Deriving cross-architecture diagnostics via Artificial Intelligence methods.** The ability to broadcast the effects of software changes to alternative CPU hardware without build tools or directed effort.

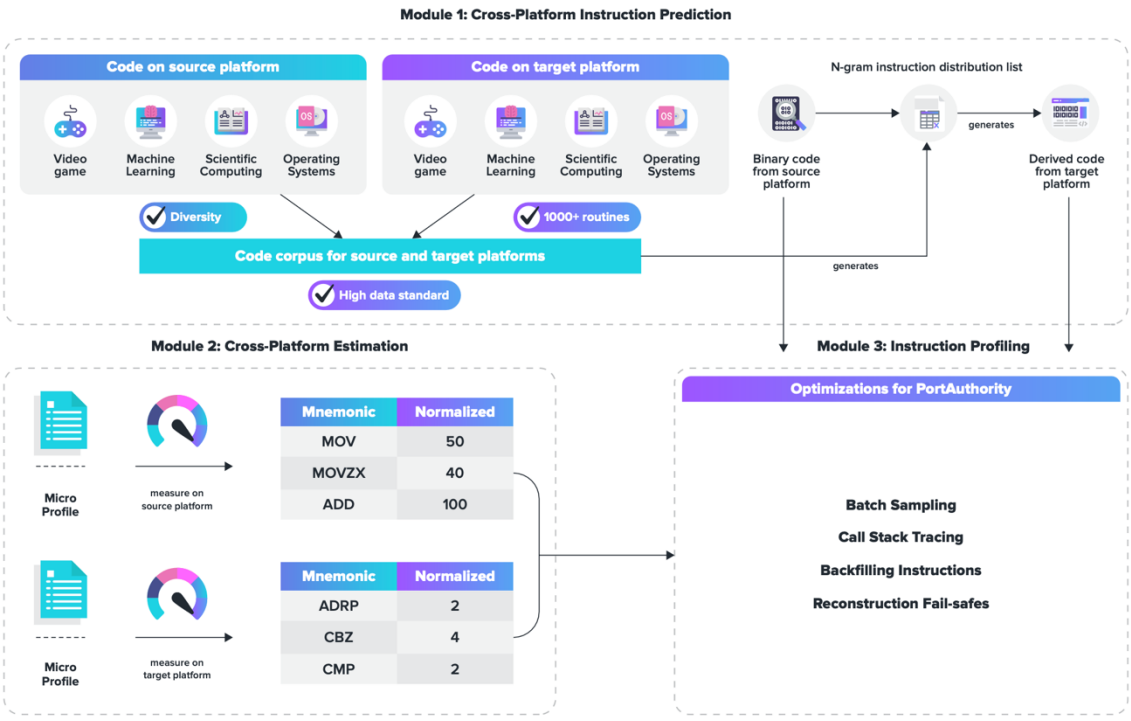

Figure 1: Overview Illustration of Proposed Framework

Methodology Overview

Figure 1 provides an overview of the proposed dynamic analysis framework, PortAuthority. First, if unavailable Module 1, instruction prediction module, uses a machine learning approach to automatically generate code for the target platform. The key component to instruction prediction is the code corpus referencing both the source and target platforms. This corpus is used to learn sufficient historical patterns of code and to map relationships between each architecture's instructions. Then using Module 2, previously recorded micro profiles are used to obtain performance data for the instructions from a given platform. Finally, in Module 3 the code is analyzed via instruction profiling and evaluation reports are generated regarding the program's structure and performance estimates for the software.

Each module is evaluated using the same set of test applications. The set includes open-source code from the video game, scientific computing, machine learning, and operating system communities. The mix of coding styles here is intended to help produce a well-rounded method capable of working with a variety of binary input. Software quality and diversity are critical factors in the evaluation of this technique. The PortAuthority tool has been tested on many additional pieces of software over the past several years and the included test catalog is representative of those outcomes.

With each test, a sample area of 2-4 million instructions is used in order to keep consistency across the recorded results. While the execution of each is highly detailed, the expected runtime of any example is only a few seconds. The executables from the test catalog are composed from 30 to 60 unique instructions per application. The overall structure of an individual program is loosely in line with the others as each generated executable is composed using identical settings and roughly matching the statistical distribution of instructions available. This emphasizes that repetitive elements involved in compiling a program may overshadow custom computation internally, especially when considering all these programs conform to the same requirements for calling functions, application binary interfaces and other forms of compliance code.

The intent here is not to displace existing tools and methodologies but to complement them. To that end, instruction metrics will be juxtaposed with conventional versions. Significant differences are debated as either advantageous or not depending upon circumstance. The insights provided by instruction metrics are divergent but also correspond closely with their better-known counterparts. Executable and Linkable Format (ELF) files can be generated for a variety of platforms. Here the work is done with what

was felt to be the two most contemporary choices for cross-architecture development, ELF files using the x86-64 and AARCH64 CPU architectures. The hardware tested included the Marcher system (Zong et. al, 2017) and the NVIDIA TX2 respectively. There is no known limitation to working with upcoming platforms like RISC-V. Conducted properly, the expectation is that the method will function well with all ELF generating targets.

The first test program is a short Fibonacci sequence calculator. Next, a slightly modified version the *logcat* application included with the Android operating system. Logcat provides a mechanism for filtering and prioritizing messages familiar to users of debug logs. Originally pulled from commit hash *cfaded* (Android Open Source Project, 2022), changes were made 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, a routine from the TensorFlow framework, an open-source platform for machine learning. This test lifts code for single-threaded matrix multiplication from version 1.4.1 and includes some infrastructure to support execution outside of the full software suite. The final test is a game project designed for the Arduboy handheld game console, Sirene. While there are several hundred games for the console, most use a single game engine. This makes results between those projects are noticeably similar. This will be shown using another Arduboy game, Shrun, much later in the paper along with a few other one-off examples to enhance experimental quality.

The rest of the paper is organized as follows. Chapter II discusses Related Work in software portability, profiling, and instruction prediction. Chapter III broadly explains the innovative, non-intrusive profiling method used in the initial research. In this chapter

a detailed use case is also presented. Chapter IV describes the basic principles behind application instruction profiling. Chapter V examines the Estimation module. Chapter VI reviews the initial work and introduces new challenges discovered while piloting the PortAuthority prototype. Chapter VII provides details related to the methodology used in the Instruction Prediction module of the proposed framework. Chapter VIII demonstrates speed improvements later added to the instruction profiling method. Chapter IX will discuss background on target application areas. Chapter X discusses future work and finally, Chapter XI concludes this study.

II. RELATED WORK

More conscious thought around future ports during the development cycle is desired. A common but poor practice is to apply improvised methods when the eventual need is discovered. Economically the costs of an improper port work against sound investment principles in a product line destined to remain on the market for a prolonged period. To accept uncertainty in new software can be uncomfortable and lack of foresight means addressing portability concerns early is an area for improvement for many developers. Software components exhibit portability when their adaptation costs are less than those of redevelopment (Mooney, 2000). This work is an attempt to improve weaknesses in the software porting process by providing programmer assistance via binary analysis. Binary analysis refers to the process of examining the raw data for a compiled application. While binary analysis is useful with or without source code it can be difficult to ascertain the key details of a program's intent when reviewing only native assembly, the given language for binary analysis. For this reason, debugging the high-level operation of an executable is more often done against the source for that program and binary analysis tends to be reserved for more specific use cases. The advantage of studying a binary directly is that it can give insight beyond what is available at the source level. Actual resource utilization for a target platform can only be known post compilation therefore binary analysis is used more often in later stages of a development lifecycle and where optimization, threat assessment, or reverse engineering are required. Static binary analysis can provide useful data without executing a program like the required space for different program segments. In contrast, dynamic binary analysis requires an actively executing program to report on a desired behavior, consider

performance counters. Each form of analysis provides access to different pieces of information, and each has notable strengths and weaknesses. Binary analysis relates closely to binary translation which in turn relates to instruction prediction. In this chapter, background for each of these areas is introduced to promote a better understanding of the PortAuthority framework.

Portable Software

Foundational work by Mooney provides a clear framework for describing various commonsense patterns to adapt software. He proposed guidelines for increasing and exploiting portability including:

- Control the interfaces
- Isolate dependencies
- Think portable

There is no quick way to predict the runtime behavior of software without executing some sample code on a device. It is unrealistic to judge software performance without a due diligence porting trial and some upfront, hand-coded work. When working with portable software an engineer needs to be confident the source can be cross-compiled and that the resulting code will run deterministically. This requires a successful cross-platform developer to keep Mooney's principles in mind constantly. If software is not portable, it could take up to a few months of redevelopment time to generate a reasonable performance estimate of the anticipated final product. Meanwhile, developing a fresh codebase is also viewed as undesirable even for a new platform. This is because

software products tend to benefit from implementation in multiple environments when they remain in the market. Thus, long life is normal for professionally developed software. Average lifetimes of ten years are common with some going far beyond that threshold (Tamai and Torimitsu, 1992). Based on these observations, it can be concluded that the likelihood of eventual porting is high (Parnas, 1994). Alternatives to porting code include forking, copying, and splitting source development in independent ways for differing environments. While considered a low-cost alternative, studies of this phenomenon have called for new techniques to automate porting tasks to reduce the maintenance costs associated with software forks (Ray and Kim, 2012).

Static Analysis

Numerous utilities are available to parse, disassemble, and even recompile existing binaries. All are forms of static analysis. The first two tasks are critical to the implementation of the PortAuthority framework, but not necessarily clearly expressed in the final user output. The ability to parse and disassemble a binary is highly valued when reverse engineering a compilation process or compiled build product. During functional development or program optimization a compiler's interpretation of a user's source can occasionally lead to interesting bugs that require visibility on the assembly level to fix (Le et al., 2015). Disassembling a binary can help diagnose this condition before software is released. In contrast, the same methods can also be used to reverse engineer a product. Reverse engineering takes place outside of the initial development cycle and potentially by an outside party of developers. The challenge is to rebuild a software product lacking its source. This process can involve stages like redocumentation and design recovery but

fundamentally must begin with the elemental assembly code due constraints viewing the original source. Recompiling working versions of existing software from reverse engineered source serves as a high accomplishment in this practice. Specific examples of tools built for this purpose are IDA Pro and Ghidra (IOActive, 2008 and Rohleder, 2019).

The Binary Analysis Platform (BAP) from Carnegie Mellon uses an intermediate representation to support program analysis (Brumley et al., 2011). The tool effectively combines the functionality of the standard GNU Binutils with a scripting language. The analytical features demonstrated by their team are static, and though not required it is also possible to execute IR created through this tool. BAP is highly extensible and tailored for the creation of custom tools built on its core methodology. Included extensions highlight a range of analysis from simple tools for translating between machine code and instruction mnemonics to more complex filters used to detect source code styles favored by the developer. BAP can also be used to build structures illustrating program execution, such as call graphs. Like many verification tools, detailed information about the run time complexity of individual functions can be derived, but information on whole program execution seems scarce. The high-level analysis provided by BAP is not suitable for evaluating the performance of an application.

Dynamic Analysis

Program optimization often requires the use of dynamic analysis, think execution profiler. While these utilities are powered by a variety of diverse techniques, binary sampling and source instrumentation are the two most common methods (Pereira et al., 2017). Sampling profilers halt and record the state of a running executable periodically.

Instrumented profilers compile in special markers that allow them to gather similar kinds of information but from predetermined locations within an application. Given this broad definition, PortAuthority could be categorized as a sampling profiler with a very short period. It is set up by default to analyze every instruction within an application for a given range. In contrast, standard profilers might only sample once every millisecond. These other profilers are suited for a much different task and usually within the scope of the same or similar hardware. Specifically, when optimizing a program, it is valuable to know areas which are frequently executed. Sampling at a high enough rate can provide a representative view of that information much faster than single stepping each instruction. Peak memory usage and other metrics can also be discovered through using interval-based profiling (Jin et al., 2012). Instrumented profilers work in a comparable way. They either log or break at distinct points within a program. This means that they depend on the processor crossing marked address locations within the program and do not arbitrarily halt the executing process. The output, however, is like that of a sampling profiler. Instrumenting an application may alter its personality and requires metadata to be included with a binary. As those characteristics work against goals for this research, PortAuthority is not implemented in a way that would allow it to be categorized as an instrumented profiler.

With twenty years of active development, Valgrind is one of the most mature dynamic analysis tools available today. The fundamental example use case for this program is to check for memory leaks. This type of validation is valuable when developing in languages without garbage collection. Though initially targeted as a memory debugging tool only, the project now serves as a generic platform for a variety of

verification tasks (Nethercote and Seward, 2007). Valgrind uses a unique just-in-time compilation technique to drive its analytical processes. A user's binary is first translated into an intermediate representation. From this state, the code may be transformed or augmented in a variety of ways before the IR is recompiled back into code for the host platform. Common augmentations for the IR include various forms of instrumentation and standard library replacement. Running the code in this modified state enables the series of extensions tied into Valgrind to perform their work with the caveats of reduced execution speed and implementation detail. For validation purposes, this approach has proved extremely useful. Valgrind is excellent for verification tasks but building an extension for this research would not be viable. As discussed with BAP, profiling the original binary representation is crucial to the usefulness of any performance output from PortAuthority. Augmentation in the form of IR obfuscates performance aspects of the compiler's intent making predictions less accurate.

There are of course informal methods of dynamic analysis as well. Lots of engineers work in an unstructured way with data directly from their development machine. Consider RAPL (Running Average Power Limit), a processor feature introduced by Intel and almost exclusively available on their chips. This feature adds registers containing continuous energy and power consumption information (Travers, 2015). Excluding previously mentioned performance counter portability problems, using RAPL is an improvement over older methods, though it still has some inherent limitations when monitoring consumption in real time. For instance, its readings are not application focused and capturing the data can incur significant overhead. Therefore, background processes using RAPL, and even direct API usage can affect the resulting

estimate (Treibig et al., 2010). Worth noting, certain other metrics are beyond the ability of a user manually monitoring performance counters entirely. PortAuthority works to eliminate both the portability and observer effect problems associated with these ad-hoc dynamic analysis methods.

Instruction Profiling

A practical application of instruction profiling is used in the open source CacheSim utility distributed by Insomniac Games (Insomniac Games, 2017). Console games are often developed on hardware more powerful than what will be available to the consumer. As such, once functional testing is complete, an optimization round will likely be required in order to ship the final product on retail hardware. Caches are orders of magnitude smaller and faster than conventional RAM, so proper usage has a large impact on performance. Thus, a frequent target of game optimization is cache utilization. CacheSim can estimate cache performance for a target architecture. The tool does this by filling a simulated version of the target cache hardware based on the record of memory accesses provided by the host. Fine grained analysis of the memory accesses within a program is facilitated by a single step debugging mechanism within the tool. Insomniac Games confirmed their usage of CacheSim as a unit test during the Xbox One console generation, though it is unclear if it is still in regular use. Noted issues with the tool include some sensitivity to array prefetching. This makes the tool overly pessimistic about cache performance in certain scenarios. There are other small caveats mentioned given that the program is not entirely hardware accurate. Those limitations will be expanded upon in the later section on instruction profiling.

Instruction Prediction

A few contemporary tools exist combining the use of Natural Language Processing (NLP) with binary analysis. Specific examples include instruction2vec and Asm2Vec, each based off the widely popular Word2Vec algorithm (Treibig et al., 2019). The focus of these two tools is currently to provide cross-architecture bug, malware, and plagiarism detection. The value in this approach is amplified by the diverse nature of assembly languages and the lack of a dictionary for direct translation. Much like a natural language, between vendors assembly languages may require multiple indirect instructions as a substitute for one from a competing architecture. Many conversational themes affect the quality of the outcomes when comparing binary forms of ported applications. To address performance, there is also a general strategy and ample prior research to expand the reach of existing data using supervised learning (Zheng et al., 2015). Lacking big data for multiple platforms extending this approach is less of an option for this study. Each of these methods utilizes machine learning for their predictions. Using the first two methods it is possible to compare program similarities cross-platform however with the latter this is not possible. Within the PortAuthority framework there is an attempt to combine the best that each of these approaches has to offer. The key difference is that PortAuthority uses machine learning to predict instructions likely to be compiled during a direct port. Performance evaluation happens indirectly later in the process and without inferencing. This means that the machine learning components of PortAuthority have more in common with the vectorization tools as there is a requirement to effectively plagiarize the innerworkings of a program for the purposes of cross-architecture performance

evaluation. These features of PortAuthority are novel and lack deeply comparable prior research.

III. NON-INSTRUSIVE SOFTWARE ENERGY ANAYLYSIS

While multitudes of tools exist to help developers improve and distribute their source code, most of these options do not operate in ways that adequately address aspects of cross-platform development. Traditionally, this is difficult because profilers assume a single target and can require other tools and libraries commonly found for that platform in order to function. There are fundamental limitations to established profiling designs where source modification, binary instrumentation, or direct access to hardware is required. For example, relying on source modifications limits a profiler's ability to target multiple languages. Virtualization is one approach that can be used to avoid these issues while profiling, but it also comes with problems of its own. Functional correctness is an excellent use case for virtualization, but also a minimum standard for quality in software. Virtualization is limited in its ability to deeply profile software as it makes abstract the innate uniqueness within a computing platform. With the availability of many correct implementations, current tools based on virtualization technology do little to directly address performance, a further level of functional betterment. This chapter outlines a study, where PortAuthority is used to attack a specific profiling problem not adequately addressed by existing tools.

An Unaddressed Problem

An IDE will regularly include features where source code improvements can be suggested or automatically applied from within the environment augmenting the skills of a developer. In addition, lower-level compiler optimizations are readily available for a variety of languages and programming setups that continue to enhance the code on the

back end. Common optimizations include transformations that automatically improve the speed or lower the memory usage of an application. However, no equivalent options exist to tailor the compilation, linking or refactoring of a program for better energy efficiency. Why is this the case? What would be required of a profiler in order to integrate energy efficiency analysis into an IDE?

Interested Parties

First, it should be established why this problem is important. Research on the popular developer community Stack Overflow has conveyed that there is growing interest from software developers in monitoring and improving the energy consumption of their applications. Unfortunately, most of the questions related to software energy efficiency are not answered or poorly answered (Pinto et al., 2014). Open-source mobile applications have been found to contain a searchable stream of energy-aware commits (Bao et al., 2016). Because developers are actively fixing consumption problems, it stands to reason that automated techniques that can help detect and locate them should also be in development.

In certain circumstances, energy aware interfaces have been created in order to capture obscure conservation knowledge and increase access to power saving techniques, removing the need for developers to sort out the best approach on their own (Moura et al., 2015). Within most projects however, bug fixes or features that require increased energy efficiency will not have the luxury of using such an interface and will need other ways to debug and eventually resolve these issues.

Existing tools such as the Battery Historian for Android can be used to debug this category of problems by reporting the active current draw from the mobile device's battery (Google, 2017). However, this tool's suggested code improvements focus on features specific to the Android platform (e.g., GPS and wake locks) in line with other reference materials from Google, restricting portability. Similarly, wisps of technical documentation from Intel delve into energy usage as it relates to algorithms and data organization but lack specific examples or toolkits that would help developers precisely replicate their results (Intel, 2011). Growing amounts of research are being conducted on the development of more aggressive and portable energy aware testing procedures due to a growing consciousness of these issues (Jabbarvand and Malek, 2017).

Solutions

Basic access to power consumption estimation on off-the-shelf development systems is normally present. Notably, software can and has been expanded with drivers connected to external meters for more accurate results, but most users will rely on the standard software implementations. PAPI is one example of a power profiling API for users of the high-performance computing languages C and Fortran (Weaver et. al, 2012). Jalen is an option for software running on the Java Virtual Machine (Noureddine et. al, 2014). Each of these approaches is backed by hardware counter registers. Counter registers can work well in the absence of physical metering as specific actions taken by the CPU greatly influence power usage. However, this is not always the case.

For many reasons, performance counters remain non-ideal for power metering purposes and different problems can arise when correlating their values to the underlying

usage. Consider the RAPL feature promoted by Intel and the challenges presented monitoring consumption in real time. Two major problems exist, its readings are not application focused and capturing the data can incur significant overhead. These limitations make it possible for background processes using RAPL, and even direct API access to affect the resulting estimate (Treibig et. al, 2010). Energy usage is inherently tied to a platform. Much more so than the other optimizations mentioned. Using less memory, or less instructions will plausibly create similar betterment cross-platform, but something like energy usage is not always tied to improved performance (Abdulsalam et al., 2015).

Existing methods to determine software energy usage require hardware support for measurement, affect run time behavior, or are tied to certain programming languages. Easy approaches are restricted, and this can lead to a lack of awareness on energy related improvements. To provide the same usability and increased code quality that are available for standard optimizations, tools need to overcome these problems and provide similar descriptive warning messages and indicators of severity. Using PortAuthority to track the energy usage of basic instructions at the binary level, it is possible to build generalized rules for improving code's energy efficiency and share that information with users at development time. The approach is non-intrusive, language agnostic, cross-platform, and most importantly can be seamlessly integrated into popular IDEs.

To accomplish this, PortAuthority can be used to build Portable Energy Scores. The Portable Energy Score is a normalized score that reflects the relative energy efficiency among basic low-level instructions such as *add*, *mov*, or stack operations. Scores are based on real power measurements from tools like the Marcher system (Zong

et. al, 2017), a fully equipped open development platform for measuring fine-grained energy consumption of arbitrary programs. As measured, instruction energy usage numbers are small, floating-point values and normalizing these to a whole number clarifies results. Once scores are established, a quality estimate for platform energy usage can be calculated using non-exact development hardware and without access to comparable measuring equipment. For example, an unmodified Intel Coffee Lake machine could estimate the relative energy usage for software to be run on Kaby Lake hardware if supplied with that platform's per instruction energy scores. Normalized instructions are added to a lookup table, and as each is encountered during the single stepped run of a program the usage estimate is monotonically increased by the corresponding energy score. Higher scores suggest more energy would be used and in what relative quantity.

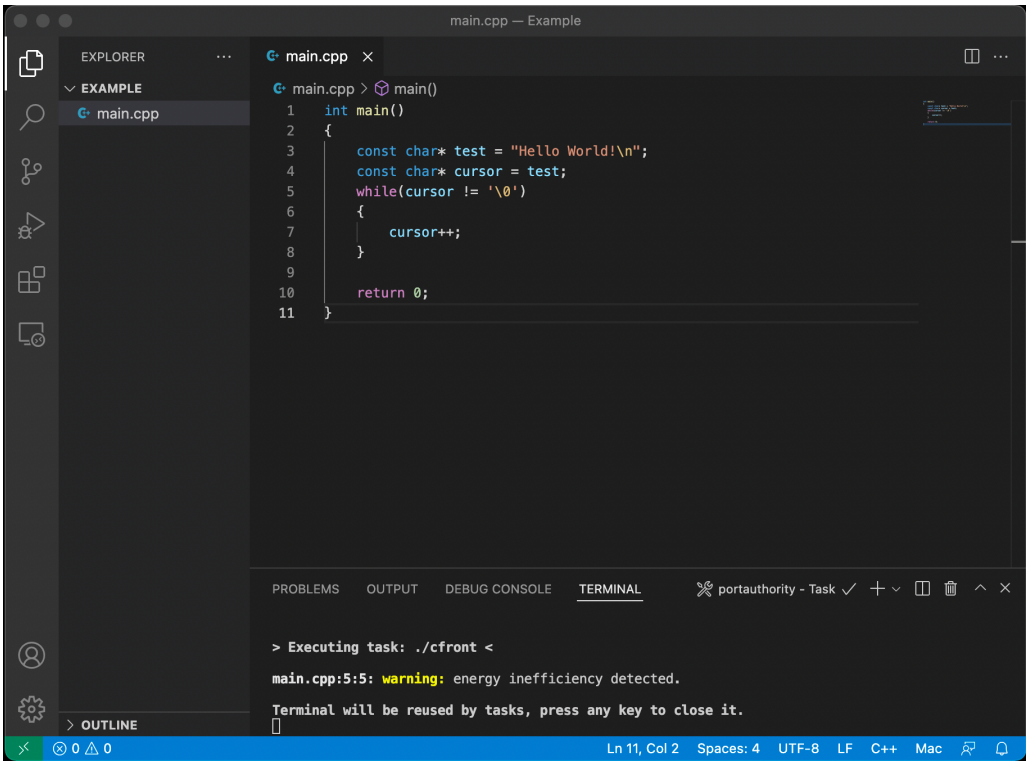

Figure 2: IDE Integration Screenshot

Figure 2 is a screenshot captured from an early IDE integration demonstration. Here Portable Energy Score analysis using PortAuthority was added to the Visual Studio Code UI and evaluated with several sample projects. In theory, the following instructions could be applied generally to any extensible IDE, though each will have some unique setup requirements. Extending Visual Studio Code with a custom integrated task can be done on a per project basis. The IDE will create a hidden folder within each project named *vscode*. This folder contains settings files for various generic IDE and debugger tasks. For this research the *tasks.json* file within this directory was modified. This file supports different task types and labels and allows users to specify specific arguments to a list of utilities. PortAuthority is a shell application but there are other supported base types including scripts. When configured correctly, *tasks.json* entries will be placed within the top level menus of the Visual Studio Code IDE under the *Run Task* option under the *Terminal* menu.

While testing this setup, three basic rules for energy efficiency were established and reported via the IDE. The first rule suggests that a developer find alternatives to replace clusters of stack instructions. The power consumption of stack instructions is significant, just below the highest of the x86-64 mnemonics tested. This could be valuable knowledge to reference during an optimization pass. Because the stack operation's Portable Energy Score is so high, it creates the potential to refactor using different instructions for better efficiency. The second rule in the set is based on the comparatively poor performance of the add instruction when compared to that of the sub operator. Their execution speed and size are similar, but power usage is substantially different. Because they are also at times easily interchangeable, finding areas to swap

higher energy usage add instructions for sub instructions makes for an attractive rule. Consider the example of a loop within a program. In many cases, processing within the loop may be order independent. This property has been exploited by other optimizations such as loop unrolling. In this instance, iterating through a loop and decrementing a control variable has shown to be more energy efficient than incrementing one. Specifically, a seven percent decrease in overall energy usage was observed over a duration of twenty minutes (Ford and Zong, 2021). The error displayed in Figure 2 was triggered by this rule. The final rule established was the doubling rule. This rule describes any section of instructions that causes register contention. Intense focus on reusing a register incurs a significant energy consumption penalty. From the study this value capped at double the measurement from a sample with properly diverse register utilization. Hence, doubling. As of this writing, energy usage evaluation as a software construct is highly unconventional. The closest work by Tumeo was intended to work on intermediate representation and did not include specific components required to produce viable estimates. (Tumeo, 2017). Rather than providing a functioning test system, this research served primarily to motivate the need for such a technology. Further details on the innovations present in this prototype using PortAuthority are detailed in the next two chapters.

IV. INSTRUCTION PROFILING MODULE

Given proper instruction information about the assembly language makeup of an application, it is possible to derive quality performance estimates for the execution of an entire program. This process trades off access to specific analytic tools for access to greater computational power. As designed the Instruction Profiling module quickly single steps an entire program and makes instruction data available to the pluggable set of analyzers in the Estimation module.

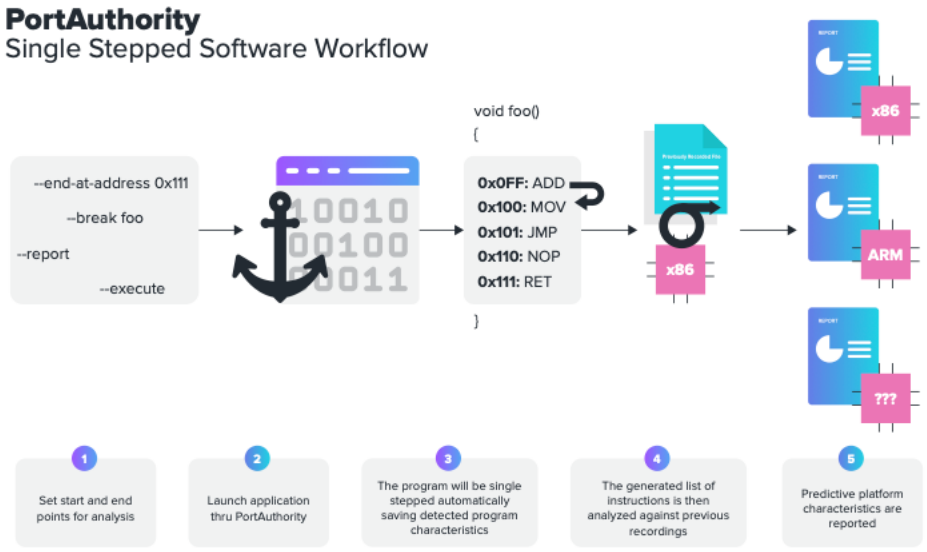

Figure 3: Single Stepped Instruction Profiling Workflow

Overview

Figure 3 illustrates the details for this module of the PortAuthority framework. First, when using PortAuthority, users have the option to select a specific profiling window based on a function signature or range of addresses. By default, the entire application is profiled. In the second stage shown in Figure 3, the module then runs the program via emulation, attaches using the GDB protocol, or where available, launches and probes the process through faster direct access using the *ptrace* call. This makes the module compatible with a large variety of processors and operating systems. To date, instruction profiles have been recorded on x86, x86-64, ARMv7a, AARCH64 and AVR architectures. During a standard profiling session, this module reads in the ELF information from an executable then steps the process to completion or within a defined boundary, as shown in stage three. PortAuthority can be extended for many forms of secondary analysis (see stage four). This is explained in greater detail in the next chapter. At the end of a profiling session, a user is given relevant console output, or a report can be generated as presented in the final, rightmost stage five. While the bulk of this research is focused on the secondary analysis provided by PortAuthority, Instruction Profiling in and of itself has analytical value. Figure 4 illustrates the categorization of the Sirene game test program. This example is faithful to the volumes of instructions expected to occur most often in the test programs. Of note, Sirene exhibits a little higher than normal percentage of arithmetic instructions and the entire test catalog is dominated by data movement.

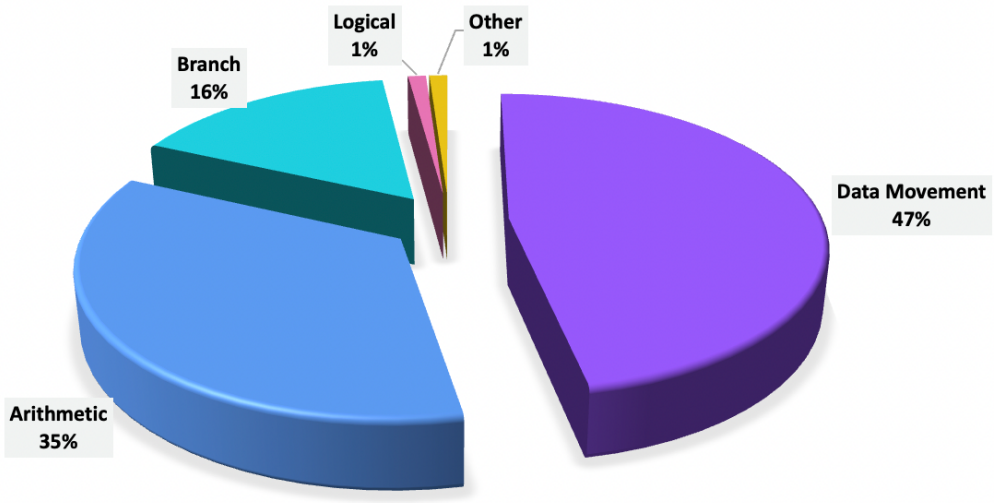

Figure 4: Instruction Categorization of Game (Sirene)

Categorization is the lens to view what exactly a binary version of a program does on hardware. Optimizations performed at the instruction level may not always align with those perceived in the source. Development environments have long provided options that help developers to optimize source code for higher performance or lower memory usage. For instance, an IDE may suggest removing or updating code. But beyond the source level, compilers like GCC and Clang also have macro-options which bundle together many smaller unique transformations into more user-friendly levels of automatically applied optimization that do not require developer input or knowledge. Other forms, like link time optimization, can be introduced even later in the build process. These options have wide ranging effects on the generated assembly code. Figure 5 shows internally how a compiler may choose to optimize code in ways not described directly via the original source.

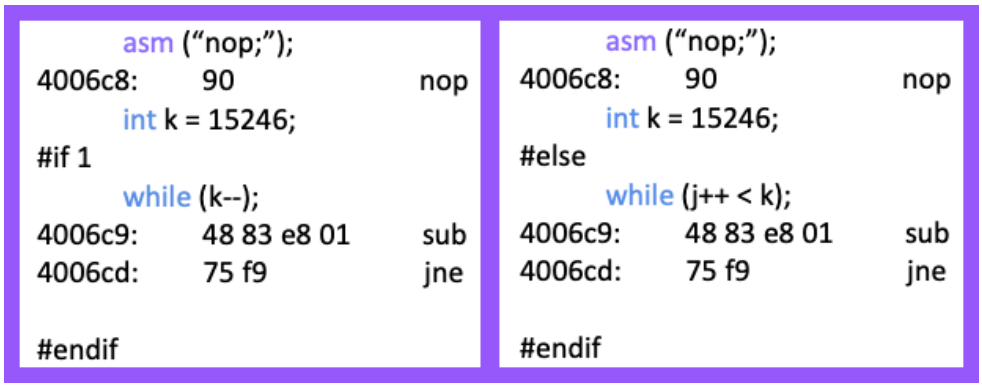

Figure 5: Example of Source Optimizations That Provide Identical Binary Paths

In the figure above, the direction of the loop makes no difference on the underlying assembly. In each version, the loop is decremented despite the developer's intent to increment the loop in the sample to the right. After a refactor the hardware version of similar source may also read quite differently than its predecessor. This indirect relationship between source code and generated instructions means that Instruction Categorization provides better insight into how source is interpreted by various compilers than higher level tools. Even working with low-level intermediate representations cannot provide the same fidelity as probing instruction outputs directly due to the post compilation optimizations injected during the build process. Amongst other benefits, in depth knowledge of the types of operations a program executes opens a conversation for developers to explore acceleration options. Floating point intensive applications may want to consider offloading CPU work to a GPGPU for instance. The tool could moreover be used to speculate on more exotic forms of acceleration like Processing-In-Memory (Ahn et al., 2015).

Evaluation

Instruction profilers reviewed and designed in this research trend pessimistic. The feedback provided by several sources has implied worst-case results are preferred in place of overly optimistic projections when profiling software. While a good measure of performance for various metrics, instruction profilers are not 100% hardware accurate and should not be considered as such. One clear oversight is the lack of consideration for out of order execution. The goal of these tools is to give user's something from almost nothing, broadly addressing the concerns of developer's preparing for upcoming and inaccessible hardware. For in depth performance analysis, other more mature tools and greater access to development hardware will need to materialize. You can see evidence of this in the results reported for the other core modules.

Validation for this module is largely defined by the other modules' results. It is possible to compare instruction output from either another stepping profiler or an emulator, but the results will lack for difference as the underlying mechanisms are the same as those implemented in PortAuthority. Comparing the differences in stepped output between specific implementations is also a dry read. As it is trivial to establish the ground truth for single stepped instruction profiling, the crucial feature to evaluate in the research related to Instruction Profiling is sampling behavior. This is the subject of a later chapter. The categorization data for each test program is listed in Table 1.

Table 1: Top Categorization of the Test Catalog

Test Program	Data Movement	Arithmetic	Branch
Fibonacci (24)	53 %	26 %	21 %
logcat	59 %	25 %	15 %
TensorFlow (MatMul)	60 %	23 %	17 %
Game (Sirene)	47 %	35 %	16 %
LINPACK	73 %	24 %	1 %

A final note on instruction profiling. It not only works to generically compare various hardware resources, but also as a way to compare different programming languages. The core test programs for this research are all based on the C language, though from various distinct sources. As a special provision for this section results are also included for a well-known Fortran benchmarking tool, LINPACK. LINPACK is a numerical linear algebra library, and a derivative is the central benchmark behind the Green500 list (Feng and Cameron, 2007). Here a historical version is used which is readily available on GitHub. Though LINPACK is written in Fortran, its post compilation structure is comparable to other ELF executables. This validates the assertion that instruction-based techniques are widely applicable and largely source agnostic for natively compiled languages. By this logic, the other modules should also be applicable to ELF programs written in languages other than C.

V. ESTIMATION MODULE

The Estimation module attaches estimates of various performance metrics to individual instructions. The expectation is that the set of instructions is provided by the Instruction Profiling module. Two techniques are employed in this research to attain the metrics tested. Other platform characteristics and extraction methods are discussed later, but here the focus is on the features which have been most robustly evaluated.

```
int i = 30000000;

int main () {
      do {
            asm("mov x0, sp");
            asm("mov x1, sp");
            asm("mov x2, sp");
            asm("mov x3, sp");
            asm("mov x0, sp");
     ... Repeats hundreds of times
       } while (i--);
}
```

Figure 6: An Example Micro Profile

Overview

A simple metric derived via instruction profiling is instruction counts. This can be used to determine the relative execution time of a program. While the static data will not give the user a measurable time to completion, the relative time difference between two programs will be obvious given that each version will have a different number of total instructions. Larger differences in instruction counts will result in larger time deltas. This is constant within a platform as the CPU instruction is the most elemental unit of

productivity with a contemporary computing device. For many reasons this may not be true if counts between two programs differ by a handful of instructions, but that vagueness dissipates quickly within the range of only a few hundred overall instances. In these exception cases, the real-world completion time would be small enough to be non-perceptible, so as a rule the method still works.

While simple instruction profiling metrics like this can be useful, if a developer desires more complex insight there are a few options to create new values. One of the established approaches to determine less apparent per instruction metrics is called micro profiling (Ford and Zong, 2021). Micro profiling requires the creation an inline assembly program containing only a representative instruction or small set of related instructions. Internally this program should loop for a period sufficient to execute several billion instruction instances. The code is referred to as a micro profile. See Figure 6. A micro profile should contain a gentle blend of slightly varied operations (like different register usage) in order provide the best outcome. While a micro profile executes, the desired metric will be evaluated via instrumentation. The measured sum of a metric for the entire micro profile is then divided by the total number of instruction instances run. This operation reveals an instruction's performance signature. Because the same instructions comprise all programs on a platform, these values can be reused when profiling other applications. Ironically, micro profiles are highly unportable between different CPU architectures, but their creation is trivial to the point of automation. Ideally manufacturers or a community of developers would also share the results of their micro profiling efforts in order to jump start the development of ports to specific platforms. As shown before,

energy usage is an example of an effective metric that can be determined via micro profiling.

Apart from measuring their effects on various subsystems, performance metrics can also be resolved by tracking the in-memory location of instruction instances. This technique is referred to here as mapping. There is a core assumption that data sizes remain similar or proportional across two platforms while mapping. Like cache utilization with CacheSim, memory usage may also be tracked using a form of instruction mapping. In this research it has been observed that decoding memory addresses resident inside certain assembly language instructions will allow those locations to be tracked as in use by a profiler. Aggregating these segments of memory during a debugging session can provide a good estimate of the maximum amount of memory required to port existing functionality across a variety of unique platforms. Any significant differences in memory usage from this estimate while profiling a future port likely indicates a programming error. For instance, while standard data types, like **int** in the C language, may compile to different sizes on separate platforms, it is not required to use these types exclusively. A portable program with vastly different data structure sizes should be refactored using more deterministic language constructs. Next, a walk through the pseudocode below to help to describe the specifics of this technique.

```
typedef value_type;

main()

    const values;

    i = 0
    j = 0
    value_type value
    value_type* test = (value_type*)malloc(values*sizeof(value_type))

    while(j < 2*values*sizeof(value_type))

        test[i++] = value
        if(i == values)
            i = 0
        j++
```

In the example above a block of memory is allocated and each value is written to twice. The exact size of the block and constant here is unimportant. To determine the memory usage of this function using instruction profiling the profiler will tag each store instruction for a given architecture. Load instructions could also be used or a mixture. Using both could provide finer results, but in most cases addresses outside the overlap in common addresses between the load and store instructions would be irrelevant. For this walk through, using the AARCH64 RISC instruction set will be comparatively brief as there are less store instructions to describe. Two specific store instructions are tracked within PortAuthority, STR and STUR. The effect of each of these instructions is to transfer data from a register to a memory address encoded within the opcode. By mapping these addresses inside an exclusive set, the profiler can discern the amount of memory used by an executable. When tracking load and store instructions relative memory access requirements can also be described. Available memory bandwidth is a key limiter to in-memory processing, the acceleration technique described previously (Ahn et al., 2015).

Accurate depictions of code coverage can also be calculated via mapping. Because this module is driven by ELF data, it is easy to know all the executable addresses within a binary. These are the instructions contained within the *.text* section of a given program. While the profiled application is executing, addresses hit during the session are captured by the PortAuthority framework. Then this module only needs to perform a simple set of calculations on these data to provide an accurate view of the percentage of an application covered. The sum of the instruction sizes at each unique address hit during the profile divided by the total size of the *.text* section is used to evaluate the instruction coverage of a particular run.

Evaluation

As mentioned, energy usage can be determined on a per instruction basis and applied generically across a set of individual programs. Prior research has shown instruction-based energy estimates for a collection of applications to be within 10% of the final values provided by other contemporary methods (Ford and Zong, 2021). Baseline measurements in previous work show PortAuthority results to not only be comparable to values provided by other software approaches (like RAPL) but also to physical measurement using specialized hardware like the Marcher system (Zong et al., 2017). The energy outcomes for the test catalog are shown in Table 2 and conform to this expectation.

Table 2: Energy Usage Comparison for the Test Catalog

Test Program	Measured x86-64	Estimated x86-64	Measured AARCH64	Estimated AARCH64
Fibonacci (48)	1615.53 J	1688.72 J	93.70 J	95.77 J
logcat	1533.32 J	1559.28 J	89.43 J	92.32 J
TF (MatMul)	3190.15 J	3233.48 J	187.50 J	190.01 J
Game (Sirene)	2709.76 J	2722.77 J	157.53 J	160.99 J

For memory usage, static elements like the data segment and program sizes can be easily determined cross-platform using ELF data. The important sections of a program all report their size in the executable's metadata. Using these data combined with the process described above to gather the dynamically allocated memory required during program execution can provide a valid impression of the memory required to load a program. For dynamic memory usage this technique was evaluated against the Valgrind *memcheck* tool. This would be roughly equivalent to the output from the *top* application on Linux or the memory usage reported from a similar system monitor if applicable for a platform. Table 3 details this research's memory usage comparisons.

Table 3: Memory Usage Comparison for the Test Catalog

Test Program	Valgrind Usage	PortAuthority Usage
Fibonacci (8)	636 B	764 B
logcat	73.26 KB	64.31 KB
TF (MatMul)	257.34 KB	258.77 KB
Game (Sirene)	69.63 KB	69.88 KB

While most of these programs have alignment across both tools, there is a material difference in the logcat values. This discrepancy highlights some dissimilarity in the information provided by Valgrind and PortAuthority. Recall that Valgrind depends on

the ability to run a program inside an instrumented run time and PortAuthority does not. The values for memory usage returned by Valgrind are based on resources allocated by the standard library. This does not guarantee that those resources are used during program execution. PortAuthority on the other hand reports exactly what pieces of the allocated memory were used during the profiled run. In the unit test for this research, logcat allocates more memory resources than it accesses. Unlike the other programs, this process has clearly identifiable unused resources. The dispute is not necessarily bad, but one metric may be more valuable than the other under certain conditions. PortAuthority's version of memory usage represents in the lowest common denominator.

Finally, coverage evaluation. As a baseline measurement for coverage quality, the Estimation module results are compared to output from another code coverage tool, Gcov. This program is a recognized standard for statement profiling, and it is included with the GNU Compiler Collection. Unlike the PortAuthority framework, to use Gcov an application must first be compiled with two flags, -fprofile-arcs and -ftest-coverage. These flags signal the compiler to instrument the source so that when run a series of log files are produced detailing the paths taken by the application. No source modifications are necessary, however creating a new binary is a required part of the multistep process to generate a clean Gcov profile. After executing the instrumented program, a set of data files are produced. These are intended to be ingested by tools like lcov which will in turn produce a human-readable coverage report. Coverage via Gcov has a healthy set of ordering and version requirements, as opposed to the non-intensive to non-existent prerequisites for PortAuthority analysis. For this research we compared

PortAuthority's instruction coverage with statement coverage from Gcov with favorable results as shown in Table 4.

Table 4: Coverage Comparison for the Test Catalog

Test Program	Gcov Line Coverage	PortAuthority Coverage
Fibonacci (24)	100.0 %	100.0 %
logcat	15.6 %	13.1%
TF (MatMul)	39.6 %	11.2%
Game (Sirene)	17.4 %	14.5%

Using DWARF data and the GNU Binutils developers can easily display source intermixed with object disassembly using *objdump*. This is the most readily available visualization of the relationship between the metrics provided by the two tools. In most cases the two will track within a narrow tolerance. From the test catalog one result stands out as obviously dissimilar between Gcov and PortAuthority. Further investigation reveals that this difference is related to the use of C++ templates. This feature allows for blocks of source code to be written using generic type information and then reused with multiple concrete data types. The feature is implemented by the compiler and results in multiple similar but distinct functions being generated in the final output binary where real types are repeatedly substituted for the generic.

In the non-conformant TensorFlow test case, templated classes are heavily utilized throughout the source. Internally to the compiler, this results in concrete functions for each method within a class being generated for each type requested by the developer. During later passes the compiler is allowed to dead strip unused code including class methods. Only the class methods necessary for proper execution are required to be placed in the final binary. However, in this example there is at least one

instance where the specific code for an unused templated method is still set to be emitted into the output binary. The compiler is reluctant to strip the function, but the reason why is unclear because rules within a compiler can be opaque. Setting the compiler options to dead strip code can increase coverage numbers using either tool, but in this instance even targeted directives to the compiler about these functions are ignored.

The presence of extra, unused instructions is a topic for debate on the meaning implied by metrics given by various tools. Coverage is frequently used by developers to evaluate the robustness of their unit test infrastructure. Specifically, it helps answer the question of whether there are enough tests to be confident in a software release. Once released, bugs may be found anywhere accessible to the user, so it is important to cover all available functions. By nature, templates are minimizing the lines of code within a source file. Executing even one variant of a template function is enough to include that line in the overall coverage metric returned by Gcov. In contrast, PortAuthority is reporting coverage based upon all execution paths generated within the binary. This is inherently a greater search area than lines of code when templates are involved. If developers are executing multiple variants in their production code, they may be lulled into a false sense of coverage when testing a single variant within a unit test framework. Using line coverage as measurement of unit test completeness does not account for templatized variants in the way PortAuthority does. However, in the research test case, the code is unexecuted which makes the value of either approach conditional.

A final note on coverage. Regardless of architecture, instruction coverage should be the same for all platforms. However, coverage percentages can drift a bit based on instruction size. Some platforms, like AARCH64, have a uniform instruction size but

notably x86-64 does not. The end calculation within PortAuthority is based on the program size in bytes, not actual instructions, so reported numbers may differ slightly based on the architecture a developer chose to profile.

VI. ARCHITECURE CHALLENGES AND PROFIING EFFICIENTLY

After living with a prototype, it is not uncommon to identify a list of immediate design improvements. Notably while the initial method developed for PortAuthority is inherently cross-platform and architecture independent it does not work cross-architecture. Profiling applications using this method requires that new potential targets are compatible with the same application binary interface as the machine used to develop the code. Instruction profiling in its initial form is valuable within a given CPU architecture across multiple distinct, conforming platforms but less so as new instruction sets are encountered. Automatic, robust cross-architecture support is critical for the adoption of any technique designed to enhance the experience developing portable code. While it is not impossible or uncommon to encounter ports destined for similarly architected or compatible systems it is also not guaranteed that this will be the case. In fact, foundational innovation in a stagnant technology market is often motivated by significant changes to a platform's underlying CPU architecture.

Another glaring issue with the initial version of the tool is profiling speed. Functionally, single stepping executable code produces the highest fidelity results when using instruction profiling. However, halting an application for analysis has a high cost with the operating systems tested. Therefore, repeated halting for every instruction executed quickly compounds the overhead, affects the debugging experience, and ultimately reduces the value of the technique in real world scenarios.

The Unaddressed Problems

The CPU architecture landscape will soon become more fragmented than it has been in a generation. This makes porting software to a new platform require considerable effort. Currently, utilizing emulation tools such as Apple's Rosetta (Apple 2020) is the most straightforward way to reuse cross-architecture software because it introduces minimal development cost. Unfortunately, emulation overwhelmingly favors compatibility over performance. In fact, degradation of 20% or more due to the overhead incurred through emulation has been reported when using Rosetta (McShan 2021). Even in the event simulators do exist, tools that can emulate cross-architecture devices for the purposes of performance modeling are lacking (Zheng et al., 2015). Creating emulators with a deep understanding of the implications of running software built for a new architecture will also lack finesse given a standard product release timeline. The presence of real hardware will always be before the completion of widely available, and highly accurate simulators. While emulation tools provide a temporary solution, their timely obsolescence is expected as a technology transition window expires. Cost effective tools that can help developers port software and avoid work arounds like emulation are becoming more valuable but there are non-trivial challenges to their development.

Adding to porting frustrations, specific hardware and supporting toolkits (e.g., libraries and compilers) may not be readily available for developers. For example, retail hardware was not obtainable until six months after Apple announced the M1 chip, the fulcrum point for their corporate shift from x86 to ARM. Even once available, the process of natively porting and creating new, dedicated code to support an alternative architecture may have a steep learning curve. The cost to fix all compatibility issues can

be high and extremely time-consuming. Incompatibility in key areas will significantly delay ports to a new platform. Despite this, development tools remain woefully unequipped to tackle multi-architecture development directly and adding yet another profiler to the mix without cross-architecture capability immediately limits its appeal.

Additionally, a hundreds-fold slowdown is observable when profiling the sample Fibonacci sequence code using the initial version of PortAuthority. While the real-world consequences for this tiny application are small, this not ideal for larger programs. When attempting to profile an application with a normal execution time of several minutes, this process quickly becomes untenable. The potential for a profiler to evaluate whole program execution is important as insights gained from profiling pieces of an application may be different than those based on running an application to completion. The market for higher precision instruments with slower speeds is clearly niche. Generally, profilers are allowed to trade some accuracy for throughput. The positive community response has continued to direct profilers to be developed in this way. The desire for lower fidelity tools that work quickly is well established and there are defined limits to expected imperfections (Patel and Rajawat, 2013). Increased speed can be accomplished through a variety of stock techniques like sampling or data filtering (Lin et. al, 2013). Time and time again researchers have affirmed the advantages of a faster profiler when compared to the hyper accurate. Eliminating redundant data or the need for verbose information is a core value in computer science. Profiling should be organized in a way where detailed information is available but also not typically required for quality insights.

With mild improvements, PortAuthority can be used in these previously problem areas to great effect. Details on the features added to PortAuthority to remedy the second

wave of challenges encountered are presented in the next two chapters. Chapter VII showcases how the first issue was resolved. It outlines the Machine Learning (ML) program used to build representative cross-architecture binaries for analysis with PortAuthority. Chapter VIII presents a modified execution sampling process whereby large instruction segments are not exhaustively single stepped. They are instead later recovered and passed to the framework as if recorded naturally. This addresses efficiency problems related to instruction profiling as the research indicates back tracing instructions is significantly faster than recording them linearly.

VII. INSTRUCTION PREDICTION MODULE

The primary responsibility of the Instruction Prediction module is to determine what cross-architecture instructions are expected to compile for a target platform based on the original instructions used on a source platform in the event of a port. This greatly extends the value of instruction profiling and estimation. Given quality micro profiles or other estimation information it is possible to create new estimates for a separate platform without officially porting software. A probabilistic n-gram model is used to facilitate instruction prediction in PortAuthority. 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 (Lee et al., 2019).

Overview

Predictive artificial intelligence in PortAuthority is powered by learned n-gram instruction distribution lists. The lists are built based on the content of a code corpus that consists of many identical functions built for each supported architecture. A small amount of compiled source data is required to reveal sufficient historical patterns of the relationship between assembly instructions from two disparate platforms. For this research, the focus is on predicting instructions in a potential AARCH64 port using a finished x86-64 source binary. To generate a distribution list, a training step is required whereby an application windows across n instructions at a time recording an x86-64 n-gram and the corresponding ARM architecture n-gram from the code corpus. Later to construct a representative ported AARCH64 version of an arbitrary x86-64 program, PortAuthority emits equivalent AARCH64 n-grams using reference n-grams from a

disassembled source application according to the learned probability from the database of previously compiled sample functions. An example of these data is shown in Table 5. This output is composed of a list of AARCH64 n-gram frequencies attached to each discovered x86-64 2-gram from the code corpus.

Table 5: N-gram Distributions For x86-64 PUSH-MOV

N-gram	Frequency %
SUB-STP	45
SUB-STR	16
STP-STR	6
MOV-STP	5
STP-MOV	5

The code corpus used in this research follows the high data standard described by Post (Post, 2018). It is critical that these executable segments are created using the same process and with identical source code across platforms. In addition to available open-source code, the corpus also includes a selection of Clang compiler tests and computer-generated C code using Csmith (Yang et al., 2011). Each function in the code corpus is compiled using the Clang compiler from the LLVM project at commit hash *bb7a57*.

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 its x86-64 counterpart. Using data from the code corpus, it can be 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 that should appear in a port. To solve this problem, the module is designed to supplement 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 corresponding platforms. Finally, all the generated instructions are passed to the Estimation module, as before with recorded instructions, for further processing.

Evaluation

The effectiveness of the Instruction Prediction module can be measured in many ways. The first perspective is context-free. Before a model generated by the module can be considered valid it must be shown comparable to a compiled application when viewed as a bag-of-words. Do the instances of various instructions closely match those of a compiled version of the same program not considering order? The test for this property is straightforward. Figure 7 depicts the expected differences between a predicted version of the Fibonacci test program before and after a complete port. The results show matching opcodes in worst-case descending order and capped at the point of insignificant difference. There are other unique instructions in each program not shown in the figure, however when judged context-free the useful differences in those instructions would not be apparent. From a very high level it is known that this program is communicating the same intent as a proper software port as it is using roughly the same instructions in roughly the same distribution. That alone is enough to enable basic instruction estimation in this area of the research.

The findings from this round of experimentation indicate that 2-gram sequencing provides the best instruction predictions. The first instruction generation scheme, shown in the top left of Figure 7, is based purely on the statistical distribution of instructions the compiler will generate for any program. All latter schemes use an x86-64 version of the core application as a source for emitting equivalent AARCH64 n-grams. The results for each scheme are shown in decreasing order of difference (left to right) versus a compiled binary. 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 prediction 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 3 and 4-gram predictions. The trend of worsening 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 bottom half of Figure 7 where 3-gram and 4-gram schemes are active.

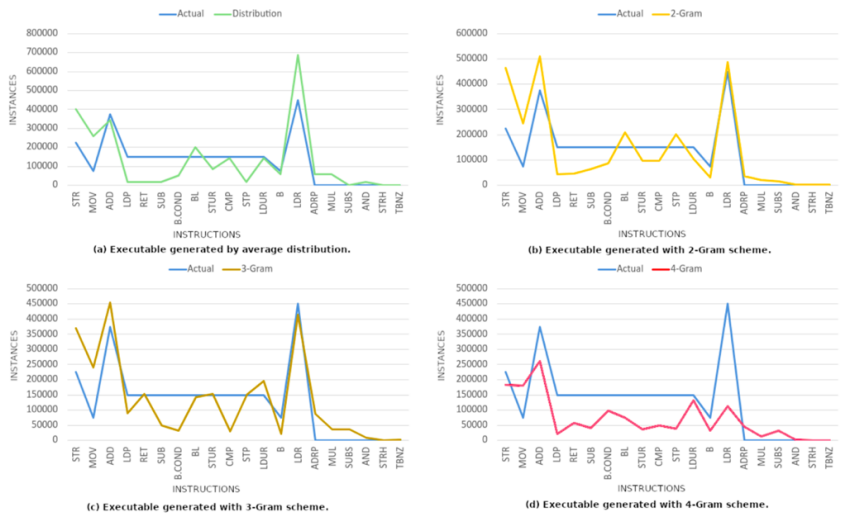

Figure 7: Comparison of Predicted Instructions to Real Using the Fibonacci Test

While context-free evaluation is valuable, applications are inherently reliant on context. A working program not only requires proper ordering, but also adherence to subtle platform and vendor binary contracts in order to execute. Cache utilization is an excellent example of why context is also valuable when estimating performance. Through experimentation, it is well known that full applications predicted by this module will have few exact instruction matches internally to their compiled counterparts. Typically, this can be as low as 8 to 10%. If you compare an instruction at an arbitrary address from the compiled application with the instruction at the same address in the predicted version, a large majority of the time it will be different. An exact match is an easy way to prove context correctness, however context integrity can be evaluated without exactly matching a reference program. This research uses the Bilingual Evaluation Understudy (BLEU) score to bridge that gap (Papineni et al., 2002).

The BLEU score 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, here it is used to evaluate the quality of code translation between different hardware architectures. The generated code for the target platform is known as the candidate. When comparing a candidate to its reference (a native port), the calculator will window over the candidate combining terms at a distance of n. In place of an exact match, scores are calculated based on instances of that n-gram in the reference code using statistical precision.

Expectations for readability can be viewed as appreciably better for each 10% improvement in a BLEU score. Scores that are less than 0.20 usually do not provide

significant value while scores above 0.60 offer the highest fluency (Lavie, 2011). The score will also penalize instructions that appear in the candidate program more times than its references. Hence there is a high level of correlation between the closest modeling executables tested using other metrics and their BLEU scores (Ford et al., 2021). See Table 6 for a full set of test results. 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.

Table 6: BLEU Scores for the Test Catalog

Test Program	Score
Fibonacci (24)	0.22
logcat	0.79
TensorFlow (MatMul)	0.33
Game (Sirene)	0.42

Based on this evaluation, 2-gram instruction prediction is again recommended 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 the tests, while a minimum BLEU score of 0.20 is maintained over all experiments. This suggests the gist of a given program is clear, albeit with significant ordering errors. These scores compliment the results presented during the context free testing. Most of the translated executables provide good, understandable translations and one test even achieves the high-quality standard.

Papineni et al. reported that natural language translations achieve their highest levels of n-gram correlation around n values of 4 (Papineni et al., 2002). Functional

blocks may contribute to the comparatively low best value in this research where n equals 2. In reviewing the mixed disassembly of the test catalog, it was observed that for some lines of code only 1 or 2 instructions are required to convey the source intent. If the majority of the functionality described in source is built using only a few instructions per line, 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. While the code corpus consisted of a thousand source functions, noticeable quality remained even with half this amount of data. Below this, experiments were less successful. To maintain this level of integrity, a code corpus must be composed from a diverse selection of software and using similar compilation processes on each platform.

Access to native cross-platform porting tools, without requirements for source code, and while lacking higher levels of interpretation is also a pioneering aspect of this research. Using the BLEU score to evaluate these ports has been well received in the low-level software community and many future research forks should be possible based on the template provided here. Simultaneous multi-platform debugging information is made possible by this creative feature and displaces other methods like static scaling or serialized multi-target testing in an expanding number of profiling use cases.

VIII. SAMPLED INSTRUCTION PROFILING

One issue often cited with the Instruction Profiling module is the lack of profiling speed. When compared with a traditional sampling profiler, following the path of each instruction as opposed to several hundred or thousands at a time has visible user impact. This research observed profiling times 200 times slower than normal program execution while single stepping, and that rate is not acceptable for analyzing software with prolonged execution time. For specific workloads, it is possible to profile a single iteration and scale instruction profiling results to minimize profiling time. PortAuthority supports scaling and the feature is adjustable to match the common limiting behaviors of executable functions. However, this does little to address problems with whole program execution as inevitably applications are composed of numerous unique algorithms, each with different Big O representations, which can be chained together unpredictably. This chapter explores additional features added to PortAuthority in order to address problems during whole program execution including batch sampling, call stack tracing, instruction backfilling, and fail-safe reconstruction. Combined these features significantly accelerate instruction profiling times with little loss in overall estimation quality for the entire lifetime of a process.

Overview

Instruction profiling has been proposed to address specific weakness of conventional development tools. In its purest form however, this method lacks the expected usability of those products. The proliferation of sample-based tools is a good indication that they typically provide what developers want from a profiler. They are

reasonably fast and can generate enough data to progress development past regular production hurdles. To bridge the gap and bring commercial viability to instruction profiling this research also includes suggestions for blending the two methodologies while retaining the best elements of both. It is possible to produce quality cross-platform instruction profiling results while sampling from a working application at the cost of increased algorithmic complexity.

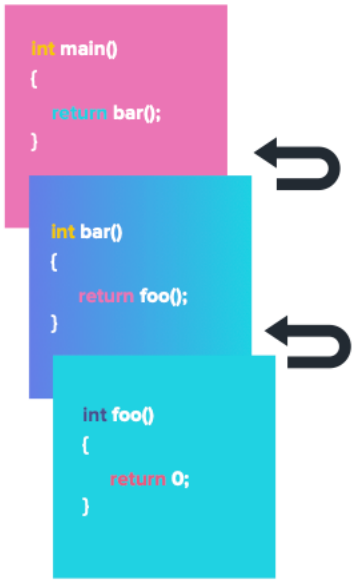

Figure 8: An Illustrated Example of Call Stack Tracing.

Instruction profiling requires a captured list of instructions near the number and categorization of an actual port to a target platform. Sampling inherently denies PortAuthority this core requirement so additional mechanisms are introduced in order to estimate the missing instruction data. The first of which is call stack tracing. Call stack tracing provides a broad overview of the functions executed over the profiling interval.

This feature is implemented in PortAuthority as a two-step process. First, the profiler will take advantage of the function call record resident in many compiled executables. Inherent to many CPU architectures is the ability to use a stack pointer register to back trace the execution of a halted, native program. This gives standardized access to the memory address of a caller from a callee in a recursive way. In a standard IDE debugger, the resulting execution chain is commonly referred to as the call stack. On conformant architectures it is then possible to navigate a list of stack pointer/instruction address pairs in applications programmatically. Because this is a property of the underlying hardware, this technique works with code built from a diverse set of source languages. Call stack tracing is feasible using any of the current PortAuthority front ends though it has only been implemented using *ptrace*. The core requirements are access to an initial value from the stack pointer register and the ability to probe an application's memory. Creative use of these features allows PortAuthority to read from the stack and enables it back trace function calls and recover instructions previously executed during the sampling interval (Figure 8 illustrates an example of call stack tracing). Each colored block in the diagram represents a function. Each function in turn calls another depicted just ahead of its caller in the Z-order of the figure. Using the method sketched it is possible to create a reversable pointer system whereby callee functions can determine their callers.

As described is the brute force approach to reconstruct a call stack. There is also a formal bytecode designed to describe all the stack decisions a compiler has made during compilation (DWARF Standards Committee, 2007). Interpretable call frame information is optionally available in ELF executables compiled with DWARF debugging symbols, but these can be stripped during production. Tailored stack data is handy when debugging

because in many languages this memory is used in an unstructured way for data beyond call frames, including local variables. The decision to crawl the stack directly in PortAuthority is driven by the desire to operate without a requirement for symbols. Using the brute force approach means that the tool can work with a greater variety of non-instrumented executables but is limited in the call depth it can uncover.

In the following experiments, while processing x86-64 ELF binaries, *ptrace* based call stack tracing is used to facilitate on-the-fly instruction prediction and generate code for AARCH64. On this profiling platform, the top of the stack always resides in the *rsp* register. Stacks by convention on Linux start high in the upper address space of a platforms's virtual memory then grow downward. To find previous frames, PortAuthority will incrementally search upwards in memory from the current frame's stack pointer to find a value greater than the initial address given by reading *rsp*. Recall the key pieces of frame information include stack pointer/instruction address pairs and that the address ranges for both the executable *.text* section and the stack are narrowly scoped. When combined with the knowledge that stack information is regularly formatted, the tool can consistently filter out stack noise and retrieve data of interest. If the search is successful and stops adjacent to a stored value that also resolves into the address range of the *.text* section of the ELF executable being profiled, the algorithm assumes it has correctly bypassed any data not related to call stack recovery. This happens recursively until PortAuthority is unable to define a valid executable frame using these parameters. Specific platforms may define the stack behavior differently, but the presence of stack pointer/instruction address pairs and some notion of exploitable compiler conventions should be commonly applicable.

Tracing the call stack identifies functions of interest but fails to present a complete set of instructions run over a sample interval. For each call discovered, only one additional correct executable instruction can be confirmed. To compliment call stack tracing another step has been added as part of the reconstruction process. This step is referred to as instruction backfilling. For each frame discovered, PortAuthority will iterate backwards from the address found in-memory by one instruction until a *ret* opcode is encountered. The next instruction is marked as the beginning of the function in the call frame being interrogated. Then the range of addresses from this boundary back to the address found on the call stack are decoded and added to the list of instructions to profile. This produces results like single stepping the process, but without the overhead of starting and stopping the process multiple times.

A *ret* instruction internal to a function would work against this scheme. Forcing an early *ret* instruction is difficult and unexpected given how compilers typically layout an execution path. Despite efforts in Figure 9 to return in multiple locations, the branches within the function all jump to the same final destination. A function has standard duties that help it prepare for its assigned tasks and to return control to the function from which it was called. Repetitive elements within a program like the prologue and epilogue make up a considerable portion of an application's execution. In fact, so much time can be lost during these boilerplate activities that there are specific optimizations to reduce their cost, like function inlining. In reference to Figure 9, each branch must execute the epilogue to conform to binary interface expectations so returning from a single location ultimately makes the most sense. While this range to *ret* search method will not always provide a perfect account of the instructions run, the behavior is reliably close to correct.

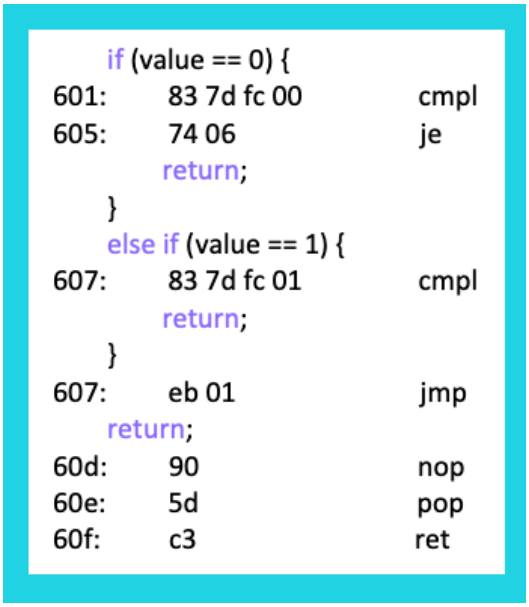

Figure 9: Placement of Compiled *ret* Instructions.

Measuring or optimizing code will provide the most benefit to a developer on longer running or more frequently hit segments. This makes sample-based profiling a good tradeoff even though it does not provide a full picture of a program's execution. However, a common complaint with this approach is that it may miss smaller sections of code that run to completion within the sampling interval. A remnant of this behavior also affects PortAuthority. When sampling, the depth to which PortAuthority can trace and backfill the call stack does not always provide a number of instructions equal to that which should run over a sampling interval. Remember that while instruction profilers are tolerant to some incorrectly reported instructions, they struggle with misreported instruction counts. Providing a statistically significant wrong volume of instructions will yield consistently worse estimates, so the tool must guarantee that somehow during reconstruction it can always provide a count of instructions near that of a single stepped

profile. Two fail-safe mechanisms were proposed during this research (statistical occurrence data and repeat until full) for use when PortAuthority establishes that a call stack is too shallow. Statistical occurrence data as defined previously is simply a continual output of probabilistic instructions from most frequently occurring on a platform to the least. The probability of an instruction is derived from the n-gram instruction prediction code corpus created for cross-architecture prediction. The repeat until full process recycles the existing data gathered from the current call stack until a collection of instructions equal to what should have been executed during the last sample interval has been accounted for. Repeat until full was found to be superior and is used exclusively in the following evaluation.

Evaluation

The familiar test programs are rerun through the previous profiling tests in order to evaluate their suitableness for sample-based profiling. In addition, extra benchmarks are added to further demonstrate expected outcomes. Batch sampling is implemented in this research by modifying the *ptrace* system call central to the operation of PortAuthority. Here the INTERRUPT parameter is used in place of SINGLESTEP. This changes the behavior of the call and allows *ptrace* to arbitrarily stop a running process and examine the current state. Looped *ptrace* calls using the INTERRUPT parameter combined with an additional call using the CONT parameter, which resumes execution from a halted state, form the algorithmic base of batch sampling for this experiment. Starting with Table 7, the categorization results for each program are recalculated using sampled profiling. In Table 8, the previously measured values are shown alongside the

newly sampled energy estimation results for the profiled x86-64 platform. AARCH64 is excluded here as the sampling technique does not change the downstream instruction prediction/profiling process outside the initial input value for the profiling host.

Table 7: Top Sampled Categorization of the Test Catalog

Test Program	Data Movement	Arithmetic	Branch
Fibonacci (24)	45 %	25 %	13 %
logcat	62 %	22 %	15 %
TensorFlow (MatMul)	61 %	24 %	4 %
Game (Sirene)	51 %	32 %	5 %
CoreMark ®	73 %	24 %	1 %
LINPACK (Fortran)	74 %	22 %	1 %
Game (Shrun)	49 %	33 %	6 %
CSmith	42 %	27 %	12 %

The fidelity of the branch instructions is the area of categorization most negatively impacted by sampling. Here a sampling interval of 4 milliseconds is used. The instruction mix of each program tends to get fuzzier with larger sampling intervals and this value should calibrated per profiling platform. As a rule, sampling will amplify the strength of more common instances within a group. Call stack tracing has been made as robust as possible in order to limit the effects of sampling, but currently only about half of the instructions are recovered. The rest of the returned data is provided via repeat until full. While not ideal, it is still possible to estimate quality energy usage results as signaled by the data in Table 8. In aggregate instruction backfilling experimentation has shown the errors recorded are acceptable as certain instruction profiling metrics are tolerant to some level of distortion.

Unfortunately, memory usage estimation is not an option while sampling. The focus for this sampling method has been instruction recovery and not register integrity.

Because the previously described method of memory usage estimation requires register data in order to map various hardware addresses actively in use, without further work to record and patch register data it is not worth attempting with this version. However, relative memory access requirements should still be possible to attain as that estimation is purely instruction based. The shallow results from live call stack tracking and the contingency of repeat until full, also distract from reasonable coverage estimations. A modified or several aggregated methods of sampling will be required in order to properly enable all described metrics simultaneously. If it is critical to analyze an extremely short segment of code, or one of the known blocked metrics the suggestion is to use the single stepping mode of PortAuthority and limit its execution range to the function(s) of interest for best performance.

Table 8: Sampled Energy Usage Comparison for the Test Catalog

Test Program	Measured x86-64	Estimated x86-64
Fibonacci (48)	1615.53 J	1582.71 J
logcat	1533.32 J	1563.98 J
TF (MatMul)	3190.15 J	3094.30 J
Game (Sirene)	2709.76 J	2681.89 J
CoreMark ®	2301.36 J	2206.34 J
LINPACK	3437.01 J	3386.27 J
Game (Shrun)	2829.87 J	2914.34 J
CSmith	1867.82 J	1912.33 J

IX. BACKGROUND

Binary Translation

For decades, the vision of allowing the same standard OS and application object code to run on different hardware platforms has existed (Altman et al., 2001). There are many advantages to allowing low-level architecture to become just another layer of software. Indeed, features like Just-in-time (JIT) compilation and research involving the Java Virtual Machine (JVM) are widespread. However, despite advantages like the reduction in porting overhead the world has yet to see many major industry players make an open virtual machine the centerpiece of their ecosystem. Google is highly reliant on their legacy Dalvik and more contemporary ART (Android Runtime), though the external adoption of these technologies is tiny in comparison to that of the JVM. The technique serves a singular purpose within their domain. Virtualization within Android allows chip vendors to easily enter Google's software ecosystem but does little to expand the reach of their respective software packages. Their primary mobile competitor, Apple, also offers some virtualization technology in the form of Rosetta 2 (Apple 2020). As mentioned above, this is intended as a bridge technology and within a few years is expected to be phased out of their core offering. High tech vendors do not like to compete based solely on the market value of a commodity.

"With industry-leading performance, powerful features, and incredible efficiency, M1 is Apple's first chip designed specifically for the Mac."

The phrase above is indicative of the market factors in the industry. Notice there is little reference to cooperation, compatibility, or emulated performance. It works to a

company's advantage to promote the custom aspects of their hardware which comes with the caveat of requiring native code. Even something as core as the width of memory addresses, integers, and other data is still flexible within the current computing landscape. Contemporary platforms are expected to support 64-bit instructions, but getting computational work done in the layers of an Edge computing system is not bound by the same convention. 32-bit MCUs are in regular usage and many companies still make 8-bit devices (Microchip, 2021). This reality should be considered at design time to enable and effectively implement a portable software component. Efficiently building software designed around a truly diverse set of hardware will require the existence of more tools like PortAuthority. Tools that report on the fundamental output of a computational device and not just in reference to other siloed products. Binary translation is a function workaround, not a final production solution for multi-platform development.

Commercial Architecture Change

Groves from IBM rightly suggested that computer architectures will always be significantly influenced by the underlying trends and capabilities of computer technologies (Groves, 2010). This has applied not only as computers transitioned from electromechanical relays to vacuum tubes to transistors to integrated circuits but could also include aspects like software. Since that writing, it has been observed that even long-established hardware concepts like Moore's Law are no longer as relevant (Hennessy and Patterson, 2018). Whereas previously the key hardware technologies that affected computer architectures were those related to density and speed of digital switches, perhaps the access time of digital storage, hardware today is starting to look very domain specific. Large processor transitions like Apple silicon can just as easily be politically or cost motivated as they are by true innovation. Today the focus of Groves' statement shifts more towards trends than capabilities. In the past notable transitions were based on obvious technical enhancements like register size (8-bit, 16-bit, 32-bit, 64-bit). However, with Apple silicon, the public interest was around bespoke enhancements to an existing processor architecture widely available to a variety of companies. For this reason, while heralded as uniquely high performing at the time devices with similar performance have quickly come to market.

If a corporation's proprietary software can be considered a significant factor in the transition from one CPU architecture to another, this leaves independent developers in a tight spot. These individuals are motivated by an entirely different set of economic, artistic, or personal concerns. For example, in the current environment developers might be disproportionately affected by foundational architecture improvements targeted at a

retail Apple product like Final Cut Pro. This realization means that software engineers need to be ready to pivot hardware at a moment's notice as near anything can force a transition.

This has been true in the video game industry throughout its history. The timeline of industry transitions can be viewed through home console generations. These periods of time outline cost competitive hardware from different organizations and adequately demonstrate the diversity that can exist in a computing field simultaneously. Developers in this industry are often faced with market concerns that are deeply divided from the business interest of the available partner gaming platforms. A recent prior generation saw Microsoft ship a console with a cost reduced PowerPC core derived from Sony's Cell processor. At that time the three biggest consoles shared enough common architecture to create applications using the lowest common denominator. The result was that the special features of each would largely be ignored (Shippy and Phipps, 2009). In contrast, contemporary games are often ported from x86 to less powerful ARM devices in order to capture the widest market available in the current console generation. Hardware transitions have happened at a quicker pace in the games industry when compared with the PC market. Entire console generations are complete within 5 or 6 years and the technology changes can stretch from almost non-existent to radical.

Conditions are right for architecture transitions to start happening faster in broader areas of the computing industry. The portability of standardized high-level languages makes this less chaotic than in the distant past but still burdensome. The content in this section bares special meaning to the author of PortAuthority who sees a

clear gap and the need for tools that transcend commercial architecture transitions like instruction profilers.

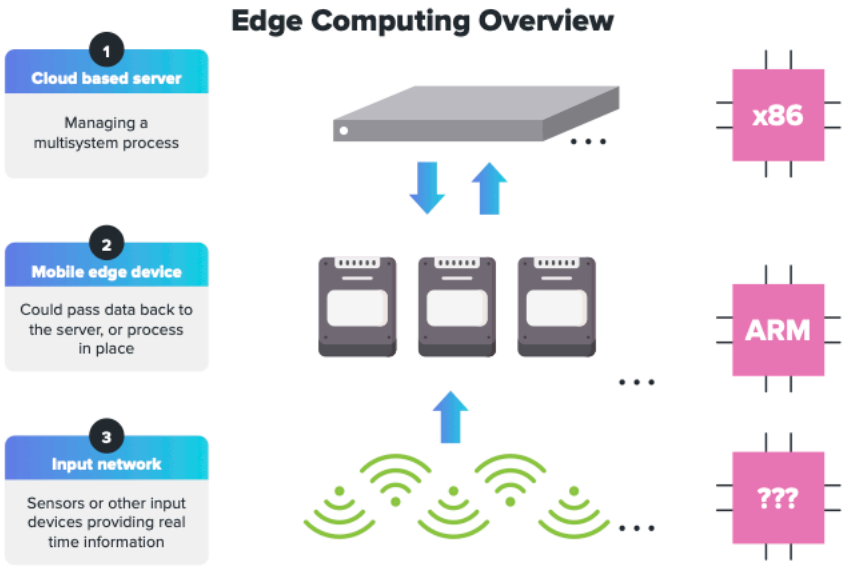

Figure 10: A Simple 3-Stage Edge Computing System.

Edge Computing

While there are many reasons to stay engaged with cross-platform development, perhaps the most demonstrative today of the power of PortAuthority analysis is Edge computing. Edge computing has clear benefits inside IoT systems that follow the pattern demonstrated in Figure 10. Specific applications include those for smart cities, Industry 4.0, and home automation. Cloud based video game streaming is another research area

where computation, specifically rendering, at the edge of a network has shown significant promise (Yates et al., 2017). Other compelling reasons to utilize Edge computing include the desire for complete offline capability. There are several consumers and geographical regions where having a fully connected system may not be wanted or possible. Safety for instance can be compromised by the overuse of network communication. Given these constraints one option is to gather and process more data near or on the generating device suggesting the use of Edge computing. These systems can be composed of several heterogeneously architected devices operating simultaneously to perform a single task. While cloud servers are currently the primary actors in IoT computation today, it is not the case that other Edge devices have little or no computational ability. Smart phones are a ubiquitous Edge computing device and excellent examples of performance on the Edge. The phone market now expects dual aperture cameras, biometric sensors, and high-resolution video to be available on most individuals. In turn, device makers are beginning to broaden their product offering to include relatively powerful devices even when directly compared to consumer PCs.

System Profilers

Figure 10 illustrates a basic, heterogenous system where it is possible to improve application performance using Edge computing as an alternative. In this research where possible the term system has been reserved for this style of multi-element application. Single computers within each system are referred to as targets or platforms. This example is a simple three-layer system containing a management server, input sensors, and smaller, mid-tier compute elements that connect the two. In the model each layer is

expected to be occupied by less hardware as the model progresses from the bottom to the top. Within this system you should expect the server to be connected via a traditional wired network. For this discussion, the next layer would be linked with a standard wireless protocol. The bottom elements by contrast can be thought of as directly connected to the middle layer edge devices. Specific speeds and technology for the network are less important when describing this scenario as proper Edge computing should benefit any combination.

As data is generated within this system, communication between the bottom and middle layer devices is expected to be near instant. Migrating data up from the mobile edge devices however requires enough time that the server's ability to process the information in real time is limited. The immediate opportunity to improve throughput in this system statically, without hardware changes, relies on processing as much data as possible on the edge devices receiving the input. Using these available compute resources reduces the burden on the application by eliminating transfers to the server, processing on the server and return trips to the mobile edge devices. None of these intermediate operations benefit the global task and are considered overhead. Implementing an Edge computing scheme in this instance would improve the system's ability to scale.

The bottom most elements in the figure could implemented as microcontrollers. For many system profilers this could represent a development problem as most profiling utilities require multitasking support and some also require access to a filesystem to generate intermediate files or test reports. This is not true of PortAuthority as demonstrated by its support of AVR processors. By design AVR microcontrollers run a single program directly on bare metal. Any requirements of an operating system or

filesystem are built directly into each application and flashed onto the unit's non-volatile memory. While excellent for embedded deployments, this limits debug access to the target. Conventional source level debugging is only available on these devices by way of a JTAG interface and a separate hardware controller. With PortAuthority however, it is possible to use emulation to gather executed instruction data via a remote gdb stub. As discussed, with a reasonably accurate list of executed instructions performance metrics evaluated via mapping to become possible instantly for any target. Provided users can create the desired micro profiling data, other discussed metrics are possible to obtain even for this much less complex architecture. For example, energy usage can be determined using physical metering on AVR via projects like PowMeter shield for the Arduino Nano (Pandauino, 2019). Real time software performance is also estimable here as instructions take well known numbers of clock cycles to execute (usually one). Using PortAuthority, most of the profiling work could be done on an established platform and extended here where conventional debugging will never be possible by design.

To reiterate, it is difficult enough to find tools and languages that span all platforms used as part of complex Edge systems. Consider also that portability is invisible when running an application and that that makes the potential for easily distributing work from the cloud non-trivial to discern. Mooney's guideline mentioned above, Think Portable, lacks some of the direct prescription of the others. He suggests not an increase in, but constant awareness of how your changes impact the portability of the design. Even if you use best practices to express portability in software, the vagueness of that standard can be limiting. The critical thinking skills involved require experience and training as well as vigilance from an engineer. Tools for dynamic analysis, performance

profiling and the like can be made available cross-platform, but the ability to use them still requires the user's code base to compile for each individual platform of interest. Even though development software many run on multiple operating systems, it can still fail to effectively aid co-design for heterogenous development teams. Many IDEs do include simulators and emulators to address co-design tasks, but they lack the ability to predict beyond functional correctness on other platforms. Few if any analysis tools are targeted towards heterogenous platform co-design like PortAuthority. The lack of automation in this debug process severely limits a developer's ability to remain aware of the portability impacts of changes they are making to their code.

X. FUTURE WORK AND DISCUSSION

The work in this research has proven itself through a few peer reviewed publications, but the PortAuthority tool itself is still primitive. There are many areas of improvement, but it is thought to be at a stage where information can be made widely available, and where others might be willing to join in the development.

Some of the immediate enhancements for the future have already been mentioned. The need for better call stack tracing with register patching is obvious. Considerable improvements here would allow coverage and memory usage estimation to be used with sampled analysis and reduce the dependency on instruction backfilling. Existing DWARF call frame data from various projects could be investigated in order to help with this task. While this research acknowledges the value of this information, PortAuthority does not use it as it is not guaranteed to be available. However, patterns buried in the DWARF data could allow for deeper and more rich call stack tracing. This change would make parts of the tool more tailored for specific supported architectures but not at the cost of the almost fully generic method available today. It could be viewed as an enhancement for select boards instead of a change for all.

Another point of interest is the greater use of intrinsic functions to gain tighter control over a sampling interval. In the latest build, PortAuthority will only use intrinsic controls for instruction counts on x86-64. Specifically fence opcodes and the *__rdtsc* call are used in the implementation. This improvement has allowed consistent, controlled samples of around 4 milliseconds, but it is worth exploring even smaller intervals. When profiling directly using an AARCH64 based target, as of now there is no equivalent intrinsic. The consequence of non-intrinsic operation is that the sampling interval must

grow substantially in order to return a consistent result using stock C++11 based timers. There are some registers of interest available on AARCH64, like the Performance Monitors Cycle Count Register (PMCCNTR_EL0), but the proper way to access these data has not been resolved. The problem might be avoided all together with a custom processor element designed to record executed instructions. While worth mentioning, that task is far beyond the capability of the author.

Of course, there is also a persistent desire to add more instruction metrics to PortAuthority. Simultaneous analysis of instruction data is core to the philosophy of the application so a new set of analyzers should not adversely affect profiling time. The most logical crop of metrics to explore next is combinatorial metrics. These would be instruction level performance metrics derived from multiple existing data points. Realistic CPU performance is an example of something that may be possible. Cache utilization is an existing instruction metric that is directly connected with reduced program execution time. Combining the worst-case performance evaluation of an instruction via micro profiling with memory usage information found through instruction mapping it should be possible to derive a CPU performance estimate closer to real world than one derived from a single metric.

Even though PortAuthority is a command line application, with a few new output options its data could be readily consumable by a conventional IDE. Something straightforward like a warning when an inquired metric reaches a defined threshold would be a good start. When properly formatted, command line can be interpreted by built-in regular expression engines within popular development environments. Modifying Port Authority's output behavior would help tap into existing user experiences for those

familiar with an environment's features. Some work has been done with PortAuthority to leverage established parsing tools within the Visual Studio Code IDE (Ford and Zong, 2021). These tools are designed to parse GCC output warnings from the command line and point back to the location within source code where the problem occurred. While preliminary work has been done in this specific IDE, there are many extensible IDEs available where the concepts tested would apply. For added effect information could also be displayed using ANSI escape codes to colorize the terminal output.

On a related note, a near future goal would be to find an appropriate software project and integrate the PortAuthority tool with a team at design time. This experience should allow for better evaluations of feature enhancements and hopefully more bug fixes. Ultimately it would be nice to have PortAuthority alongside a project for a few development cycles in order to create a post-mortem analysis. The feedback expected on the tool includes topics from usability to value. Everything learned here could be reincorporated back into future versions of PortAuthority.

Also worth investigating are other methods for instruction prediction. As shown in Table 5, the most common 2-gram match used for PUSH-MOV instructions occurs 45% of the time. Afterwards there is a steep taper across the frequency of the other AARCH64 choices. Within the research corpus there are 2-gram matches with perfect 1:1 translation between the two architectures, however the regular best returns are lower than 50% and paired with several less used combinations like the values in Table 5. While the current method has yielded positive results, the process would be improved by bettering these numbers and lowering the number of 2-gram combinations. One possible improvement would be to consider the previous state when selecting an appropriate pair

instead of the raw probability. A second pass that accounts for previous instructions may improve this piece of PortAuthority.

There is a reoccurring feature request to make generated PortAuthority binaries conventionally executable. Unfortunately, no obvious path to create a fully working, automatically ported application exists. It is possible to build an ELF file that will run in a limited capacity from the generated list of instructions, though with considerable caveats. For one, no method has been devised to break up the monotonous block of instructions resulting from the code generation process into something more akin to standard functions. The only known way to process the ELF is linearly top to bottom with no branches or jumps. Structurally the entire executable path becomes like an unrolled loop in the resulting binary. Additionally, the output of the code is nonsense. The ELF will run uninitialized and lacking proper data, variables, etc. As a bag of instructions used for the purposes of instruction profiling the "compiled" content here works fine. In comparison to a standard binary, it will be large, unreadable, unmodifiable and produce worthless results. These limitations make the value of such a binary questionable for most tasks, however it would make a great experimental control group and demo if the additional work was made to have these false ELF files execute normally.

The procedure to produce a false ELF was first described in a related work (Ford et al., 2021). Before a significant refactor, PortAuthority could only read and operate on ELF files. To get estimation data during this phase of the program's development the researcher used *objcopy* to inject raw binary data into incomplete ELF files using the *update-section* option. This tactic allows developers to override the text section from a basic C program with generated instruction data. The file is still not executable on its own

due to limitations patching faults caused by ABI non-compliance, branch endpoint changes, and stack corruption. Next, a virtual machine called the Flat VM is used to facilitate execution. The Flat VM contains shared micro profiling data from the core of PortAuthority. For each clock of the virtual machine, it increments by one instruction and applies the appropriate estimation metric like the single stepped PortAuthority workflow. With improved instruction generation this requirement could be omitted.

Most of the previous method's steps can be retained when creating a properly executable binary. The largest difference takes place during instruction generation. The goal is to mimic the Flat VM while only relying on the ISA. Greater knowledge and tailoring of instructions will be required at code generation time to pull this off. Most instructions can operate on a single unchanging state and not cause breaking runtime errors. For example, the *mov* instruction. If this operates on two registers, it is likely that by finding registers that are not a part of the ABI and then using those as part of every generated call will succeed unilaterally. Other instructions, like those that access memory, would need very specific updates in order to keep execution flowing strictly from the current address to the instruction at the next address from beginning to end. Access to many sections of memory is often limited by the application runtime. Each instruction that accesses memory would need to be patched dynamically in order to ensure it touches an area that will not trigger exceptions by the CPU. Another obvious problem is with branch instructions. These would need to be patched to always route to the instruction one address ahead of the current location. Other issues are expected, but each should be addressable within the context of the ISA provided generated instructions are emitted with a greater level of detail.

The final piece of near future PortAuthority work is related to interpreted language support. There is a path for ELF files to give targeted debug information based on lines in the developer's source. It is based on elective DWARF information being compiled into the program. If the profiler is injected into earlier phases of a project, it should be possible to reduce some of the data austerity PortAuthority is designed to content with and make a mixed conventional/instruction-based tool that gives even higher quality insight on a code bases' relatively portability. Extending this feature to interpreted languages is more difficult however because there is no obvious connection from the runtime to the developer's application source. In preparation for this research PortAuthority was tested on code written in Python, PHP, and JavaScript. This technique could also be used for other languages like C#. When launched, PortAuthority was able attach to their respective interpreters and report on the relative nature of each program but correlating runtime behavior to source changes was overly difficult. In early tests, trivial examples could be fully described using the PH7 PHP interpreter. The Hermes JavaScript engine was also trialed. There needs to be additional research into a glue layer. This feature would require some implementation per language, and that work was found to be outside the scope of this project.

XI. CONCLUSION

This work has been difficult and requires a set of knowledge and skills not widely available. Working at the machine and assembly language levels to describe in detail actions beyond functional programming is non-trivial. Instruction profiling becomes essential only when developers find themselves without the comforts of years of instrumentation and tailoring for established platforms. PortAuthority is a tool for the harshest environments.

Some would believe that this work is fully unnecessary. While its value may be limited for short spans of time during a dominant platform's heyday, a remnant of developers will always need something like PortAuthority. The success of instruction profiling will be determined by whether there is constant improvement on the design during the off season. Rushing to complete a tool like this as the need arises ignores the inherent challenges. To tie back into the future work discussion an unexpected but perfect use case for a beta version of PortAuthority must be present amid the current chip shortage. Thousands of manufacturers are scrambling to build their products around silicon supply chains that no longer exist. Only companies with the capability to reengineer their hardware and quickly port their software have a path to continue forward during this unprecedented time. With PortAuthority, it is possible to study an existing piece of software from a supply chain-stricken product and give reasonable direction as to what could be substituted.

Capability with high powered silicon is a requirement, but the long tail of value for PortAuthority may be in the static technology marketplace. MOS 6502 style processors used in the earliest PCs are still widely used in embedded products. Code

analysis and substitutions can be more easily solved in this category and the volumes of new units still produced are significant. Based on availability or for cost savings many product owners might consider backporting. How far back could the PortAuthority method effectively be backported? The 6502 is likely too far, because the architecture's owner has not progressed the compiler to modern standards. Other owners, like those for the AVR architecture, have made a conformant compiler available for their 8-bit CPUs. At a minimum PortAuthority requires an ELF binary meeting the Unix System 4.0 standard. Temporally this puts the foundation around the year 1987 during the inaugural release of GCC which targeted devices like the Motorola 68000. The ELF binary design has had far more viability than any of the compliant CPU architectures and for now still seem less volatile. That is the basis for the decision to make ELF the operational core of PortAuthority. There has been consideration given to mainstream software platforms like Windows and OSX that do not natively use the ELF executable format. While it may not be possible to run PortAuthority to profile directly on these platforms, it is possible to drive the profiler using micro profiling data recorded under those operating systems. The Clang compiler used in this research equally targets ELF and the specific formats of these vendors, so there is a bit of flexibility even outside the preferred ELF requirement mentioned here. The relative work to parse instructions from Windows executable or Mach-O files is possible but with compiler equality unnecessary.

It's certainly unfair to talk about viability without threats to viability. The first would be a sudden rise in non-assembly language-based platforms. There are boards designed to immediately run interpreted languages like Python. In the past similar efforts like Jazelle, branded Java native processors, found only limited success but markets can

change. As they exist today, these products run small VMs and are driven by processors that still rely on instructions. However, hardware could be written to work on the Python language directly and in that area of the market PortAuthority as designed would be much less useful. A similar problem occurs with Quantum computing. OpenQASM and other assembly languages for these processing elements bear little resemblance to those for von Neuman or Harvard architectures (McCaskey and Nguyen 2021). Again, in this space the work happening inside PortAuthority would need to be thought about much differently.

This project has typically been thought about as something that promotes operation on existing or upcoming hardware, but it could also be used to help design hardware from scratch. Assuming the fictional hardware stays within the lanes described by the previous paragraph, a user could extract the operations most required by their applications and graft those requirements on to a new, yet unspecified design. This takes the previous point on co-design to the next level. Taken to the extreme, a version of this tool could be used to generate ideas for an ASIC that could increase algorithmic performance.

Ambitions aside, as the tool works today it is suggested that PortAuthority run on the strongest hardware in a supported fleet. Instruction profiling is CPU intensive, and the best hope to reduce that burden, sampling, has proven difficult to calibrate. Sampling is a feature that was introduced later in the development of PortAuthority so thoughts on how to achieve the best balance of speed and accuracy are comparatively immature. The aim of this researcher is to push for improvements in cross-platform development, not to provide a perfect solution. To say the PortAuthority process is easier than a conventional process would be wrong, though complimentary feels right. If this project promotes

others to attack portability during software development in an equally creative fashion, it will have met its intended return. Perhaps that is a challenge to anyone to show better simultaneous, multi-platform, debug information to a developer. Researchers need to get others thinking about this problem in a different way to truly innovate. Promote being the operative word; to further, advance, raise.

The predominant open question on instruction profiling is what other metrics are possible? It's unclear, but theoretically the data for most anything should be baked into the executable. Even non-direct CPU metrics like network bandwidth should be possible. If accepted that this information should be resident in a compiled program, then the question is more how to efficiently and accurately mine it.

REFERENCES

Abdulsalam, S., Zong, Z., Gu, Q., Qiu, M. (2015). Using the Greenup, Powerup, and Speedup Metrics to Evaluate Software Energy Efficiency. *In Proceedings of the Sixth International Green and Sustainable Computing Conference – IGSC '15,* pages 1-8.

Altman E. R., Ebcioglu K., Gschwind M., Sathaye S. (2001). Advances and Future Challenges in Binary Translation and Optimization. *Proceedings of the IEEE*, pages 1710-1722.

Ahn, J., Hong, S., Yoo, S., Mutlu, O., and Choi, K. (2015). A Scalable Processing-In-Memory Accelerator for Parallel Graph Processing. *In Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pages 105–117.

Android Open Source Project. (2022). Platform Core Repository. https://android.googlesource.com/platform/system/core/

Apple Inc. (2020). About the Rosetta Translation Environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

Bacchus, A. (2021). Microsoft goes all-in on Windows 10 on ARM. https://www.digitaltrends.com/computing/microsoft-goes-all-in-on-arm-build-2021.

Bao, L., Lo, D., Xia, X., Wang, X., Tian, C. (2016). How Android App Developers Manage Power Consumption? An Empirical Study by Mining Power Management Commits. *IEEE/ACM 13th Working Conference on Mining Software Repositories*, pages 37-48.

Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J. (2011). Bap: A Binary Analysis Platform. *Computer Aided Verification,* pages 463-469.

Ding, S. H. H., Fung B. C. M., Charland, P. (2019). Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. *2019 IEEE Symposium on Security and Privacy*, pages 472-489.

DWARF Standards Committee. (2007). DWARF Debugging Format Version 5 Standard. http://dwarfstd.org/Dwarf5Std.php.

Feng, W. and Cameron, K. (2007). The Green500 List: Encouraging Sustainable Supercomputing. *Computer*, Volume 40, pages. 50-55.

Ford, B., Zong, Z. (2021). PortAuthority: Integrating Energy Efficiency Analysis into Cross-Platform Development Cycles via Dynamic Program Analysis. *Sustainable Computing: Informatics and Systems*, Volume 30, page 100530.

Ford, B., Qasem, A., Tesˇic´, J., Zong, Z. (2021). Migrating Software from x86 to ARM Architecture: An Instruction Prediction Approach. *2021 IEEE International Conference on Networking, Architecture and Storage - NAS 2021.*

Google. (2017). Battery Historian. https://github.com/google/battery-historian.

Groves, D. (2010). Brief History of Computer Architecture Evolution and Future Trends.

Henderson, J. (1988). Software portability. Aldershot, Hants, England: Gower Technical Press

Hennessy, J.L., Patterson, D. A. (2018). A New Golden Age for Computer Architecture: Domain-specific Hardware/Software Co-design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture - ISCA 2018*, pages. 27-29.

IOActive, Inc. (2008). Reverse Engineering Code with IDA Pro (1st. ed.). Syngress Publishing.

Insomniac Games. (2017). CacheSim. https://github.com/InsomniacGames/ig-cachesim

Intel. (2011). https://developer.android.com/training/monitoring- device-state/index.html.

Jabbarvand, R., Malek, S. (2017). Advancing Energy Testing of Mobile Applications. *In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion*, pages 491-492.

Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S. (2012). Understanding and Detecting Real-World Performance Bugs. *SIGPLAN Notices 47*, pages 77–88.

Lardinois, F. (2020). Apple announces the M1, the first chip in its Apple silicon family. https://techcrunch.com/2020/11/10/apple-announces-the-m1-the-first-member-of-its-apple-silicon-family.

Lavie, A. (2011) Evaluating the Output of Machine Translation Systems. https://www.cs.cmu.edu/ alavie/Presentations/MT- Evaluation-MT-Summit-Tutorial-19Sep11.pdf

Le, V., Sun, C., Su, Z. (2015). Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. *SIGPLAN Notices 50*, pages 386–399.

Lee Y., Kwon H., Choi, S.-H., Lim, S.-H., Baek, S. H., Park K.-W. (2019). Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN. *Applied Sciences*, Volume 9.

Lin, L., Liao, X., Jin, H., Li, P. (2019). Computation offloading toward edge computing. *Proceedings of the IEEE 107*, 1584–1607. https://ieeexplore.ieee.org/document/8758310.

Lin, Y. D., Ho, C. Y., Lai, Y. C., Du, T. H., & Chang, S. L. (2013). Booting, Browsing and Streaming Time Profiling, and Bottleneck Analysis on Android-based Systems. *Journal of Network and Computer Applications 36*, pages 1208-1218.

McCaskey, A. and Nguyen, T. (2021). A MLIR Dialect for Quantum Assembly Languages. *Proceedings of the International Conference on Quantum Computing and Engineering - QCE '21,* pages 255-264.

McShan, F., 2021. Performance of Rosetta 2 on Apple M1. https://mjtsai.com/blog/2020/11/16/performance-of-rosetta-2-on-apple-m1.

Microchip Technology Inc., (2021). Microcontrollers and Microprocessors. https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors

Mooney, J. (2000). Bringing Portability to the Software Process.

Mooney, J. (2004). Developing Portable Software. *Information Technology*, pages 55-84.

Moura, I., Pinto, G., Ebert, F., Castor, F. (2015). Mining Energy-Aware Commits. *In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 56-67.

Nethercote, N., Seward, J. (2007). Valgrind. *SIGPLAN Notices 42*, pages 89–100.

Noureddine, A., Rouvoy, R., Seinturier, L. (2014). Unit Testing of Energy Consumption of Software Libraries. *In Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1200-1205.

Pandauino. (2019). PowMeter for Nano. https://pandauino.com/en/powmeter-for-nano.

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J. (2002). Bleu: A Method for Automatic Evaluation of Machine Translation. *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Parnas, D. L. (1994). Software aging. *Proceedings of the 16th International Conference on Software Engineering - ICSE '94,* pages 279-287.

Patel, R., Rajawat, A. (2013). Recent Trends in Embedded System Software Performance Estimation. *Design Automation for Embedded Systems 17,* pages 193-213.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J. (2017). Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering - SLE '17*, pages 256– 267.

Pinto, G., Castor, F., Liu, Y. D. (2014). Mining Questions About Software Energy Consumption. *In Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22-31.

Post, M. (2018). A Call for Clarity in Reporting BLEU Scores. *Proceedings of the Third Conference on Machine Translation.* Association for Computational Linguistics, pages 186–191.

Ray, B., Kim, M. (2012). A Case Study of Cross-System Porting in Forked Projects, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12,* pages 1–11.

Rohleder, R. (2019). Hands-on Ghidra - A Tutorial about the Software Reverse Engineering Framework. *Proceedings of the 3rd ACM Workshop on Software Protection - SPRO '19,* pages. 77–78.

Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal 3*, 637–646.

Shippy, David and Phipps, Mickie. 2009. The Race for a New Game Machine: Creating the Chips Inside the XBox 360 and the Playstation 3. *Citadel*.

Tamai, T., Torimitsu, Y. (1992). Software Lifetime and its Evolution Process Over Generations. *Proceedings of the Conference on Software Maintenance – ICSM '92,* pages 63–69.

Tumeo, A. (2017). Architecture Independent Integrated Early Performance and Energy Estimation. *Proceedings of the Eighth International Green and Sustainable Computing Conference - IGSC '17,* pages 1-6.

Travers, M. (2015). CPU Power Consumption Experiments and Results Analysis of Intel i7-4820K. Newcastle University. http://async.org.uk/tech-reports/NCL-EEE-MICRO-TR-2015-197.pdf

Treibig, J., Hager, G., Wellein, G. (2010). Likwid: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. *Proceedings of the 39th International Conference on Parallel Processing Workshops – ICPP 2010*, pages 207–216.

Weaver, V. M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., Moore, S. (2012). Measuring Energy and Power with PAPI. *Proceedings of the 41st International Conference on Parallel Processing Workshops,* pages 262-268.

Yang, X., Chen, Y., Eide, E., Regehr, J. (2011). Finding and Understanding Bugs in C Compilers. *SIGPLAN Notices 46*, pages 283–294.

Yates, R.D., Tavan, M., Hu, Y., Raychaudhuri, D. (2017). Timely Cloud Gaming. *Proceedings of the IEEE Conference on Computer Communications - INFOCOM '17*, pages 1-9.

Zheng, X., Ravikumar, P., John, L.K., Gerstlauer, A. (2015). Learning-Based Analytical Cross-Platform Performance Prediction. *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation - SAMOS '15,* pages 52–59.

Zong, Z., Ge, R., and Gu, Q. (2017). Marcher: A Heterogeneous System Supporting Energy-Aware High-Performance Computing and Big Data Analytics. *Big data Research 8*, pages 27-38.