UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

Studiu de Caz al Evaluării Costurilor de Portare: Portarea IxOS pe Plăci ARM

Lucian-Ioan Popescu

Coordonatori științifici:

Dr. ing. Lucian Mogoșanu
Conf. Dr. ing. Adrian-Răzvan Deaconescu

BUCUREȘTI
2022

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

Case Study of Evaluating Porting Costs: Porting IxOS on ARM Boards

Lucian-Ioan Popescu

Thesis advisors:

Dr. ing. Lucian Mogoșanu
Conf. Dr. ing. Adrian-Răzvan Deaconescu

BUCHAREST
2022

Contents

SINOPSIS

Portarea software e costisitoare și consumatoare de timp. Refolosirea software-ului a devenit o practică standard în ingineria software datorită beneficilor sale de salvare a costurilor, astfel e important să înțelegem cadrul de cuantificare al costurilor de refolosire astfel încât procesul să poată fi optimizat. Ca să explorăm această problematică, luăm ca exemplu experiența noastră de portare a infrastructurii de testare Ixia pe un sistem off-the-shelf, popular, cu suport pentru Linux, cum ar fi Raspberry Pi și extragem costurile de portare asociate cu acest proces. Acest lucru ne ajută să creăm un model revizuit al portării pe baza căruia extragem costurile de portare pentru proiectul nostru. Mai mult de atât, prezentăm factorii care au afectat portarea și corespondența lor directă cu modelul de portare și costurile asociate. Descoperim că procesul de portare nu e liniar dependent de gradul de portabilitate al sistemului, acesta depinde foarte mult de familiaritatea dezvoltatorului cu sistemul. În sfârșit, discutăm limitările modelului nostru de portare și facem o comparație cu modelul vechi de portare.

ABSTRACT

The software porting process is costly and time consuming. Software reuse has become a standard practice in software engineering due to its cost saving benefits, therefore it is important to understand the framework for quantifying the costs of reuse so that the process can be optimized. In order to explore this issue, we take our experience of porting the Ixia network testing infrastructure on a popular, off-the-shelf system that supports Linux, such as Raspberry Pi and extract the porting costs associated with this process. This leads us to a revised model of porting based on which we extract the costs for our project. Furthermore we present the factors that affect the porting and their direct correspondence with the porting model and costs. We discover that the porting process is not linearly dependent on the degree of portability of the system, it very much depends on the familiarity of the developer with the system. Finally, we discuss the limitations of our porting model and make a comparison with the old porting model.

THANKS

I would like to thank to my coordinators, Lucian Mogoșanu and Răzvan Deaconescu, for their guidance and implication. Also, I would like to show my gratitude to Adrian Dobrică, Elena Mihăilescu and Darius Mihai for their advices and suggestions. In addition, I would like to give thanks to Iustin Bîliș for his editorial review.

Chapter 1

Introduction

Porting a software system and evaluating the porting costs is a hard problem in systems programming. [1, 2, 7–11, 13, 15–17] The reasons for porting software systems are various: the developers want to enhance the performance, the hardware environment starts to get deprecated or the system wants to take advantage of features unavailable in the current environment.

Software development costs and invested resources for understanding, maintaining and developing new features for a system are big [3, 14, 19]. People would like to preserve these investments when the need for a new environment arises. To achieve this goal, people designed programming languages and compilers that increased portability, operating systems that could run on multiple hardware platforms [9] and various standards that allowed developers to talk to the computer using well defined interfaces [18]. However, the porting process is a non-trivial endeavor up to this day, it is error prone and time consuming.

As with other software development processes, porting has its own costs associated. It is important to evaluate them and understand their implications in the project so that the developer can optimize the process of porting in the future and call attention to the weaknesses and strengths of the process. In this work we describe the experience of porting the IxOS testing infrastructure, used by Ixia for high performance network testing, on ARM off-the-shelf boards. We do this for two reasons. Firstly, we are interested in exploring Ixia testing infrastructure on newer environments with the hope that we will improve the performance and reduce the costs of delivering the system. Secondly, we are interested in evaluating porting costs related to the project using the Ixia software, which is known to be industry-grade and versatile, and thus being sufficiently complex as a practical case study.

On one hand, we make the following contributions in this work: we review the porting models described in [8, 11, 16] and propose a more general model that can be applied to modern software porting, we review the porting costs factors described in [8] and analyze their relevance in our porting work, and we provide guidelines and discussions on the topic of improving software porting based on the experience of porting IxOS infrastructure. On the other hand,

we port the testing infrastructure on ARM arhitecture. During this process we isolate the relevant parts of our porting in a separate repository. As a result, the isolated components can now be used independently from IxOS by internal teams interested in further developing of testing tools and systems on ARM boards.

This thesis is structured as follows. In Chapter 2 we present various terms associated with porting: what porting is, porting models, porting costs and factors. Chapter 3 introduces the revised porting model that we use for our cost extraction. In Chapter 4 we present the architecture of the system and divide the porting work in multiple steps. For each step we present a description, the targets and milestones, and the final results. After the porting process is discussed, we highlight in Chapter 5 the porting costs associated with our work, including an analysis of the factors that influenced the presented costs. In Chapter 6 we conduct a discussion about the porting process and the porting costs where we investigate porting difficulties, observations about the porting tasks and ways to improve the porting process. Finally, in Chapter 7 we present the conclusions of the porting processs and its cost evaluation, and set the enhancements and objectives for further work.

Chapter 2

The anatomy of the porting process

The process of reusing code in a new environment has clear advantages over rewriting [7]. However this process does not come cheap, as the task of reusing code through porting is time consuming. In this chapter we present the theoretical background and the related work in the field of porting including the porting process with its tasks, porting costs and porting factors.

2.1 Porting and porting tasks

Porting is the act of producing an executable version of a software unit or system in a new environment based on an existing version [12]. This is seldom an easy task because, in general, it involves a good amount of code refactoring and rewriting. It can be avoided, however, if, in particular, the original design has portability incorporated by using, for example, constructs as multi-platform libraries, modular code or standard compiler behaviors [17].

The environment is defined as the set of software and hardware elements that interact with the system. This includes, but it is not restricted to: operating system, communication methods, configuration files and system variables, hardware architecture or human interaction. Two software environments involved in the porting process will never be the same because of the software and hardware inconsistencies. On one hand, programs between operating systems will not work, even if the hardware is the same. For example MacOS uses MACH for executable files while Linux uses ELF, moreover even if they follow the POSIX standard, they may have implementation details that do not align with each other. On the other hand, processor architectures vary from one another in the way they understand machine language and even if they do not vary, custom hardware attached to these processors may make porting difficult as the software must be rewritten in order to accomodate the new peripherals.

Mooney [12] presents two components of the porting process: transporation and adaptation. The first is described as the act of moving the system (code or binary executable) to a

new environment and the latter is described as the act of modifying the system in order to be compatible with the new environment. Transportation is facilitated by communication channels to the target environment, either online (file sharing systems, remote connections) or physical (using data storage devices). Adaptation consumes more development resources than transportation because it implies translating the source code to the new environment, solving possible inconsistencies and making sure that the software system behaves in a well defined manner when ran in the new environment.

Mooney's model is very simplistic regarding the tasks that can occur during a porting process. Hakuta and Ohminami [8], and Tanaka et al. [16] created a more accurate model that reflects better components involved in porting an application. The tasks involved in their model are the following:

- Advance preparations
  - Surveying development environment
  - Surveying OS
  - Surveying program for porting
  - Surveying workstation development environment
  - Adjusting target environment
  - Initial source code modifications
- Workstation testing
  - Standalone testing on workstation
  - Linked testing on workstation
- Target testing
  - File-making
  - File system creation
  - Installation on target
  - Test program creation
  - Linked test on target
- General duties
  - Documentation
  - Progress tracking
  - Discussions

They emphasize on spending additional time on getting familiar with the system and only then starting to port the application per se. Testing is conducted in a workstation environment (that is, testing the builds on the local machine or in a local simulator/emulator) and in the target environment. Finally they also include non-technical duties as documentation, tracking and discussions.

2.2 Porting costs and factors

Porting costs, and more generally, software development costs, are measured in man-hours [8, 16]. While the costs are determined by program size and contents [8], other factors as portability impediments, human factors or environmental factors [8] play a considerable role. To understand the factors that influence the porting costs, these factors are quantified in indices that describe how much of an influence they have.

In our work we will use three indices of this type as follows: portability impediments index, human factors index and environmental factors index.

The first index, portability impediments, answers the following question: how portable was the program to be ported and how many difficulties did the developer meet in the porting process? The factors that influence this index are described in (S1∼S11 [8]) and the index is computed using Equation 1.

$$\alpha_p = \eta * \sum_{n=1}^{12} \omega_i S_i \tag{1}$$

Here η is a portability design index, ω_i is the weight assigned to each factor and S_i is 1 when the impediment factor i exists, otherwise is 0. The factors are placed in three categories: differences in processor architecture, OS disparity and differences in language processor.

The second index, human factors, answers the following question: what role did the experience and knowledge of the developer play in the porting process? The factors that influence this index are described in (H1∼H5 [8]) and the index is computed using Equation 3.

$$\sum_{n=1}^{5} H_i \tag{2}$$

Here H_i are the human factors presented in [8]. Their values range from -2 which reflects the maximum productivity while 2 reflects the minimum productivity.

The third index, environmental factors, answers the following question: how did the development and testing environments, and the tools used during the porting process affect the porting costs? The factors that influence this index are described in (E1∼E3 [8]) and the index is computed using Equation 3.

$$\sum_{n=1}^{3} E_i \tag{3}$$

Here E_i are the following environmental factors as presented in [8]:

- Development environment (E1)
- Unit test environment (E2)
- System test environment (E3)

As for H1∼H5, the values for E1∼E3 range between -2 and 2, -2 being the best score for E_i, while 2 being the worst.

After we described the anatomy of the porting process and presented the porting costs associated with this process, we present a practical example of porting a software system and evaluating the costs.

Chapter 3

A revised porting model

Given that the model of porting presented in Chapter 2 was crafted for the particular use case of one project [16] and we wanted to use a more general porting model, we revised the old porting model and modified it so that it could match more porting projects.

In the revised model, we keep the *General duties* and *Advance preparations* tasks and modify the *Workstation testing* and *Target testing*. We do these changes because we want to emhpasize the allocated time between testing and development with the *Building for target environment* and *Testing* tasks.

Following is the revised model:

- Advance preparations
  - Surveying development environment
  - Surveying target OS
  - Surveying program for porting
  - Surveying documentation
  - Adjusting development environment
  - Adjusting target environment
  - Initial source code modifications
- Building for target environment
  - Build system triggering and modification
  - Installation on remote environment
  - Reviewing inconsistencies between source and remote environments
  - Solving problems with external dependencies
- Testing
  - Testing in simulated environment
  - Testing in target environment
- General duties

- Documentation
- Progress tracking
- Discussions

In *Advance preparations* the developer familiarizes with the tools, environments and the program to be ported, and also adjusts the development and target environments for creating and testing the program to be ported. Finally, if needed, the developer also makes *Initial source code modifications* that reflect the modification of the source environment to the new target environment (e.g., modification in system call numbers and error numbers [4]).

In the second task of *Building for target environment* the developer focuses on three issues: compiling the code to generate binaries for the target environment, installing the code in the target environment and solving the inconsistencies between the source and target environment.

The previous task and *Testing* are the core of the porting process, they deliver the ported application that operates in the target environment. Testing is of two types in this model. The application can either be tested in a simulated environment for convenience (e.g., hardware is not available at the moment of testing) or it can be tested directly in the target environment. Furthermore this tasks includes implicitly the time allocated for setting up the testing environment.

The last task, that is *General duties*, encompasses subtasks related to human interaction activities. In this part of the project the developer focuses on delivering documents that describe the process of porting or other information relevant to the project and focuses on planning and discussing aspects with regards to difficulties encountered during the process.

Chapter 4

Porting IxOS on ARM Boards

To continue our study of evaluating the porting costs, we chose to port a large scale system used for network testing. In this section we present the process of porting this system.

4.1 Porting Architecture

The components of the porting architecture, presented in Figure 1, are:

- A client for printing network testing data (**IxExplorer**)
- A middleware for connecting the client to the machines that run the network testing suite (**Chassis**)
- The actual machines that run the testing (**Cards**)
- The device that benefits from network testing (**Device Under Test** - DUT)

IeExplorer is a Windows application that allows the user to connect to the machines that run the testing, configure and run tests on them, and retrieve information about the status of the tests.

To allow IxExplorer to easily connect to multiple machines that run the testing, a Chassis is used. The most important application that runs on the Chassis is IxServer. It creates a communication channel between IxExplorer and the testing machines.

Next, the cards are the most interesting part for our porting. They run on a custom Ixia solution, IxVM to achieve the best performance for network testing. On top of IxVM is placed IxOS Linux, a modified version of Linux with custom device drivers, kernel parameters and userspace applications. In terms of userspace applications, InterfaceManager and IxStack offer the control plane traffic generation mechanisms for network testing. They are the two most interesting applications for our porting are InterfaceManager and IxStack. Their role in the system is to manage the network interfaces, represented by Port01 and Port02 in Figure 1, and to load the network protocols for the testing suites.

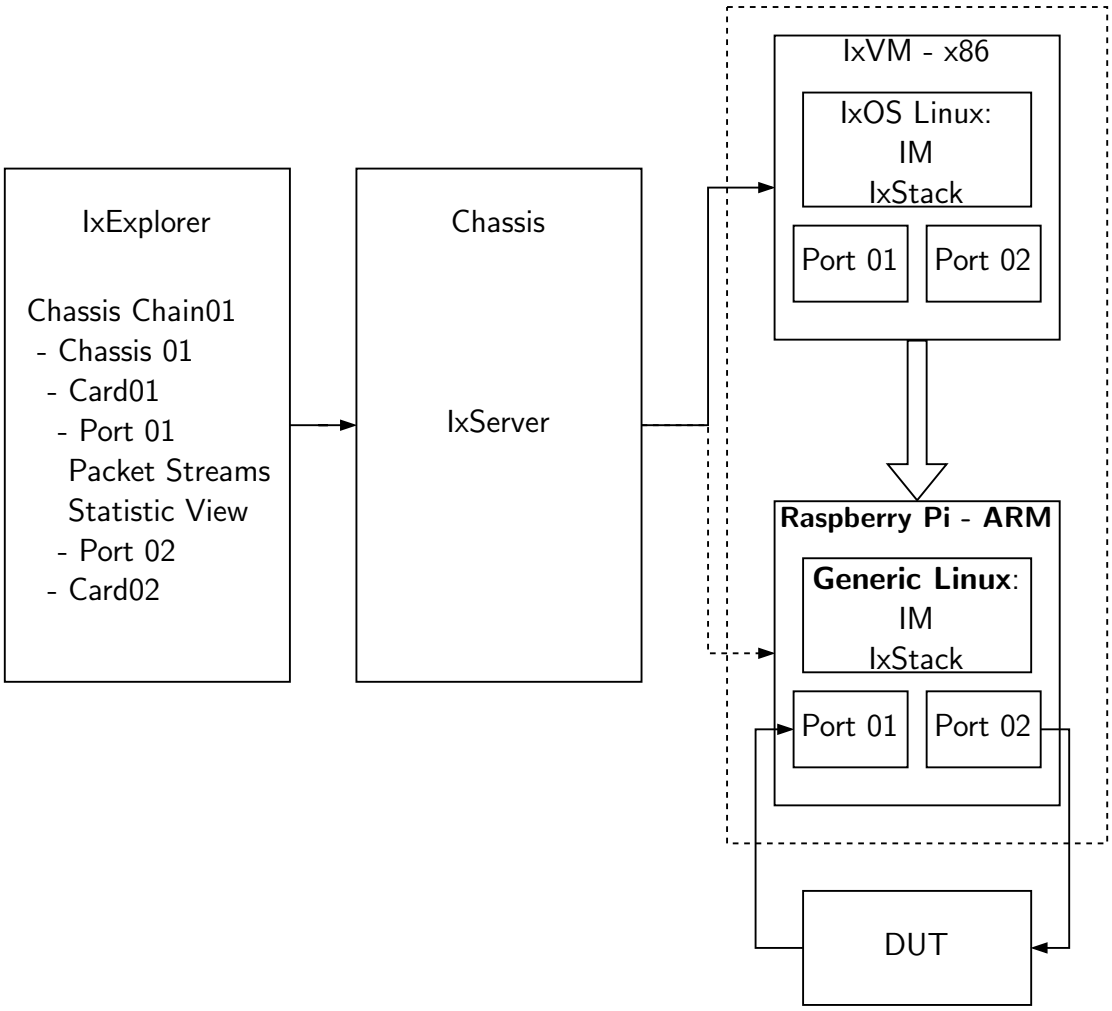

Figure 1: IxOS porting architecture. The right part of the figure presents the modification of the card environment. IM=InterfaceManager

The device device under test is the consumer of the network traffic produced by the cards. Usually it is connected between two **Ports** that will monitor the behavior of the DUT while receiving traffic.

4.2 New IxOS Architecture

In Figure 1 we highlight the changes we made to the initial architecture. First, we changed the hardware a card runs on. From IxVM we moved to a Raspberry Pi 4. On top of the Raspberry Pi, we install a Raspberry Pi OS Linux instead of IxOS Linux.

These changes required us to port InterfaceManager and IxStack to the new Raspberry Pi environment so that we could benefit from the same functionality as in the old Ixia custom hardware + IxOS Linux environment. To recreate the required environment for InterfaceManager and IxServer, we had to install custom tools, various types of pipes, shared libraries and configuration scripts in the target environment.The process of porting the two applications is

described in the next section.

4.3 Previous work in this area

Before we started the project, there was an attempt to make InterfaceManager and other applications independent of IxOS. The previous project focused on compiling the whole IxOS infrastructure for x86 and then extract the binaries for InterfaceManager and other relevant applications.

This helped us during our porting because the result of the previous work was a portable system that could easily be moved in another Linux environment. Making InterfaceManager independent of IxOS meant that all the assumptions made by the application regarding the operating system interface were removed (e.g., custom device drivers and proc entries) and the work of porting it to another environment was simply a task of finding the right tools for building the binaries and solving unknown inconsistencies.

InterfaceManager had already a high degree of portability as it was written in C++ using an object oriented paradigm. The architecture dependent code was separated using constructs as ifdefs and the coding style was compiler agnostic, meaning that we were able at any time to plug another compiler and generate the correct binaries.

4.4 The Porting Process

The porting process consisted in moving the IxOS testing infrastructure from its source environment to a new target environment consisting of an off-the-shelf ARM board, Raspberry Pi, running a stock Linux operating system. We chose an ARM board because we wanted to run the legacy infrastructure on in a new environment compatible with the old one.

When we started the porting we set the following porting milestones:

- Decouple InterfaceManager build from legacy IxOS
- Run InterfaceManager in QEMU
- Run InterfaceManager on Raspberry Pi hardware

We achieved all our proposed milestones during the twelve weeks of the project. At the end of the project we divided the work into three logical stages that cover the process of achieving the above milestones.

The target of the porting process was to decouple InterfaceManager build from IxOS and run it on an ARM off-the-shelf board. By doing this we also wanted that the Raspberry Pi to be visible from IxExplorer as a card. Because of this we needed to maintain compatibility between

the chassis and our Raspberry Pi card. We achieved this by porting another component from IxOS (i.e., HostProxy) which is responsible for communicating directly with the chassis.

The logical stages of our porting process are the following:

- Build binaries for ARM64
  - Separate InterfaceManager + HostProxy from IxOS infrastructure
  - Integrate IxStack with InterfaceManager
  - Build plugins for InterfaceManager
  - Test initial builds in QEMU
- Create Card environment on RPi
  - Run setup on RPi hardware
  - Modify InterfaceManager to run in the new environment
  - Debug InterfaceManager initialization issues
- Bring up ports on Chassis
  - Find the cause of the InterfaceManager initialization issues
  - Solve the problems regarding the link state of the card

These stages with their respective substages were not completed in chronological order. The process of completion was rather incremental, meaning that for example we had to test the initial builds in QEMU each time we made a modification in the *Integrate IxStack in InterfaceManager* stage, going back and forth between the two substages.

In the first stage, we started the work of porting by trying to separate the relevant components from IxOS and build them for ARM64. We started with InterfaceManager and HostProxy (proxy for communication with IxServer), which were the building blocks of our porting, and continued with IxStack and its components. The hardware was not available at this time so we tested our builds using an emulator for the target architecture (i.e., QEMU).

Next, when the hardware arrived, we used the Raspberry Pi to test our builds. This proved to be a great advantage for us because the testing environment worked better on hardware than in the emulator. After we built all our components, we started to focus on the inconsistencies between the source environment and the target environment. For that we had to modify some parts of the program that were source environment dependent. In this process we had problems with the initialization of InterfaceManager on the Card. This required an interative process of testing, searching, rebuilding and redeployment to understand and solve the initialization problems. The major inconveniences with the initialization featured missing utilities as ethtool(8), missing configuration files and missing code for ARM architecture. Another problem was the inability to use the same tools on the target environment as on the source environment. On Raspberry Pi OS we were not able to use dmidecode(8) because the operating system did not allow us to access /dev/kmem even if we switched to user root.

In stage three we investigated the initialization problems from another perspective. This time we investigated what was the initialization sequence on IxServer. Here we discovered that IxServer did not set the link state correctly when connecting to our Card. To solve that we tracked the variables and code zones that modify the link state in order to find out which code lines cause trouble. In the end we discovered that a problem regarding the initialization of the protocol loader was causing our start-up issues. After we solved these issues we were able to fully port InterfaceManager on Raspberry Pi and integrate it with the bigger system consisting of IxServer and IxExplorer.

Chapter 5

Evaluation of Porting Costs

In this section we present the costs associated with the porting process described in Section 4. We divide our work in tasks that can be individually evaluated based on the revised model discussed in Chapter 3 and describe the methodology of extracting the porting costs based on the progress tracking we have done during the project.

Later we discuss the factors that affected our cost evaluation. We focus on three aspects of these factors: portability impediments, human experience and environmental factors. For each of them we analyze a score to understand their impact in the project and we analyze them with regards to our porting.

5.1 Costs of Porting IxOS on ARM Boards

To evaluate the costs for porting IxOS infrastructure we divided our work based on the model described in Section 2. Using the progress tracking done during the project, we extracted the time spent on each task. We use man-hours to evaluate the cost for each task. The results of this process is shown in Table 1.

The most time consuming task is *Build system triggering and modification*. The reason behind this is spending a lot of time on extracting and integrating IxOS components as InterfaceManager, HostProxy and IxServer. There was also much repetitive time spent on triggering the compilation pipeline that was added in the total of 67 hours.

We spent an approximately 2.5x more time on testing than on solving errors and inconsistencies. This means that the errors were not difficult to solve, instead they were difficult to find. This situation is no surprise for us because we had no reference or documentation that would help us in investigating the errors and inconsistencies.

Porting task	Subtasks	Man -hours	Subtotals /subtask (%)	Subtotals /task (%)
Advance preparations	Surveying development environment	15	4.16	14.35
	Surveying target OS	5	1.38	
	Surveying program for porting	3	0.83	
	Surveying documentation	5	1.38	
	Adjusting development environment	10	2.77	
	Adjusting target environment	13.8	3.83	
	Initial source code modifications	0	0	
Building for target environment	Build system triggering and modification	67.56	18.76	32.71
	Installation on remote environment	15.26	4.23	
	Reviewing inconsistencies between source and remote environments	23.03	6.39	
	Solving problems with external dependencies	12	3.33	
Testing	Testing in simulated environment	33.35	9.26	24.53
	Testing in target environment	55	15.27	
General duties	Documentation	30	8.33	28.26
	Progress tracking	12	3.33	
	Discussions	60	16.66	
Total		360	100	100

Table 1: Man-hours evaluation for porting tasks

There was no time allocated on making initial source code modifications because the system was unfamiliar and hard to understand from the start. Finally, we spent more time on *General Duties* than on *Testing*. This shows that there was a substantial effort put in trying to make use of discussions to clarify the system. In the end we suceeded in porting the system, meaning that the discussions we had helped us to clarify the various parts of the system that were not understood in the beginning.

5.2 Methodology of extracting costs of porting

To extract the costs of porting we used the progress tracking created during the 12 weeks of porting. Each week we allocated one hour to discuss the current status of the project and the items we plan to do in the following week. A sample of our tracking looks as following:

```
# 02.08 - 06.08

Status
  * IM is booting, having config issues
  * created RPi setup
  * deployed RPi + vChassis + IxExplorer setup
    * port is visible from IxE but we have issues with link state
      * maybe there is a problem with published stats
  * attempted connections from vChassis to vCard
  * ran into configuration issues
  * need to trick the vChassis into beliving that we're on x64

Planning
  * fix port down/hwfault report from IM
    * trick the chassis into beliving we're legit IxVM
    * make HostProxy - chassis connection
```

During a week we allocated 30 hours for porting. 5 hours were allocated to daily meetings were we discussed the plan for the respective day and 1 hour a week was allocated to progress tracking, resulting in 24 hours of work for solving technical problems related to porting. To convert the weekly tasks recorded in the tracking in the effective man-hours presented in Table 1, we matched each porting task on the progress tracking task. At first, the total amount of 24 hours is divided equally between the porting tasks, following adjustments based on factors as difficulty and frequency.

Let us take the above example to extract the porting tasks from the progress tracking tasks.

- IM is booting, having config issues :

  - Build system trigging and modifications
  - Installation on remote environment
  - Reviewing inconsistencies between source and remote environments
  - Testing in simulated environment
- created RPi setup
  - Adjusting target environment
- deployed RPi + vChassis + IxExplorer setup, port is visible from IxE but we have issues with link state, maybe there is a problem with published stats
  - Adjusting target environment
  - Reviewing inconsistencies between source and remote environments
- attempted connections from vChassis to vCard
  - Testing in target environment
- ran into configuration issues
  - Testing in target environment
  - Reviewing inconsistencies between source and remote environments
- need to trick the vChassis into beliving that we're on x64
  - Reviewing inconsistencies between source and remote environments

We have six different porting tasks that we must allocate time to. Each task has 4.16 hours initially. However, *Build system triggering and modification* did not take this long, neither did *Installatino on remote environment*. These are task that will have less time allocated than the initial 4.16 hours. We will have 2 hours for Build system triggering and modifications and 1 hour for Installation on remote environment. The remaining 4.32 hours will be allocated evenly to the remaining four tasks. Finally we have the following results for the week:

- Advance preparations
  - Adjusting target environment: **5.24 hours**
- Building for target environment
  - Build system triggering and modification: **2 hours**
  - Installation on remote environment: **1 hour**
  - Reviewing inconsistencies between source and remote environments: **5.24 hours**
- Testing
  - Testing in simulated environment: **5.24 hours**
  - Testing in target environment: **5.24 hours**
- General duties
  - Progress tracking: **1 hour**
  - Discussions: **5 hours**

After this algorithm is applied on each week we get the results in Table 1.

5.3 Factors of porting costs

While the costs of porting, in our case man-hours, are determined directly by program size and content, other factors and impediments as human experience and environment disparities must be taken into consideration.

To determine the impact some of these factors had on our porting process, we use the indices described in Hakuta and Ohminami [8] that give a quantitative influence of porting factors and impediments.

5.3.1 Portability Impediment Index

The first index that we compute is the portability impediment index. In our case eta has a value of 2, meaning that "the non-portable parts of the program are not localized, but the correspondence of program codes to their functions is clarified". For simplicity we will assume that ω_i is 0 if the impediment was insignificant, 0.5 if the impediment had a normal difficulty and 1 if the impediment was hard to solve. In our porting we discovered the following portability impediments:

- Difference in compiler specification (S8)
- Scope of library support (S9)
- Implementation-dependent libraries (S10)
- Difference in operating system interface (S12)

It can be noted that we added an additional impediment nonexistent in Hakuta and Ohminami's list [8], namely S12, which can be placed in the "OS disparity" category. This impediment is reflected in the usage of dmidecode(8) on Raspberry Pi Linux versus on x86 Linux. On Raspberry Pi we were not allowed to access /dev/kmem which was needed by dmidecode(8), which in turn was needed by InterfaceManager.

Given these impediments, the portability impediment index has the value calculated in Equation 4.

$$\alpha_p = 2 * (0.5 * S8 + 1 * S9 + 1 * S10 + 0.5 * S12) = 6 \tag{4}$$

There were no differences between processor architecture (S1∼S5), little difference between source and target OSes (S6, S7, S12) and a major difference in language processor (S8∼S11). The portability difficulty is thus reflected in Figure 2.

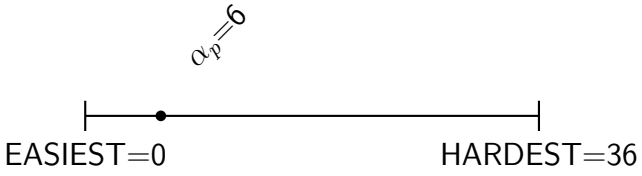

Figure 2: Portability Impediment Index

5.3.2 Human Factors Index

Software programs are byproducts of human activities that incorporate our problem-solving capabilities, cognitive aspects and social interaction [5], therefore it is vital to understand what role did the human factors play in our porting so that we can assess the quality of the porting process.

The value of the human factor index in our case is computed in Eqution 5.

$$\alpha_h = H1 + H2 + H3 + H4 + H5 = 2 + (-1) + 2 + 1 + 2 = 6 \tag{5}$$

This score is very close to the worst possible score as seen in Figure 3.

Figure 3: Human Factors Index

Indeed, the human factor played a major role in our porting process, let us analyze each factor and see what went wrong.

Knowledge about the program to be ported functions and structures (H1) As this was the first interaction with the program to be ported it was hard to familiarize with its functions and structures, therefore we had no experience whatsoever with the use cases of the program. This resulted in the inability to easily solve the porting problems that appeared in our way.

Knowledge about the hardware and OS of the target system (H2) Hardware disparities were not a big concern in our porting as we used a portable operating system and programming language that abstracted out hardware problems. Therefore the focus was on the target operating system. Our porting moved the program from an older version of Linux to

a newer one. This helped us as we did not have to learn the peculiarities of another operating system so that we could finish our porting. However there were some problems that we faced regarding the target operating system that we solved relatively straight forward.

Knowledge and experience in the area of software porting (H3) As the experience of software porting was lacking there were many situations and problems that could have been solved better, for example: doing better tracking of the porting process, putting more effort in technical discussions in order to better understand the problems, etc.

Knowledge of and experience with the language and program to be ported in use(H4) The experience with the programming language (C++) helped us to grow the productivity of the porting process as we did not have to care about low-level impediments as endianness or data alignment. However we did have problems with the language specification between different compilation toolchains that costed us a whole week to solve. Furthermore, the experience with use cases of the program to be ported was lacking.

Knowledge about the functions and usage of tools used in the development and testing environment (H5) While working in the development and testing environments we used a considerable number of technologies, some of them being either new or not trivial to work with. Here is a short list of these technologies: QEMU networking, SCons build system, Perforce versioning system and internal tools as packaging system. This meant that we had to allocate additional time to ramp-up with each of this tools after continuing with our goal.

5.3.3 Environmental Factors Index

The tools used in the development environment and the testing mechanisms used in the testing environment also add their bit in the porting costs.

In the development environment we had no documentation describing the compiling and linking procedures and no documentation regarding the environment dependent components that we need to modify in order to move the code to a new environment. We modified the code and the build system by trial and error. This reduces the score for E1 as this strategy might not always work or it could take an unsatisfactory amount of time for large and complicated systems. However we managed to understand the peculiarities of the development environment and get the first builds available for testing in two or three weeks, which is less than 25% of our porting time.

In the testing environment we had test programs and tools available such as emulators and debuggers, furthermore the file transfer and conversion tools were available from the start of the project. This facilitated the testing of our program to be ported by allowing us to focus on the porting inconsistencies rather than on testing infrastructure. Things were not perfect

however, we did have problems with the testing infrastructure that we solved in a short period of time.

The environmental factors index is computed in Equation 6.

$$\alpha_e = E1 + E2 + E3 = 0 + N/A + -1 = -1 \tag{6}$$

e are in the satisfactory half of the index, meaning that even if we had some difficulties setting and understanding the development and testing environment, we managed to work with them in order to achieve our goals. The score is also described in Figure 4.

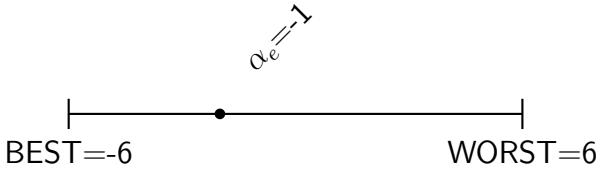

Figure 4: Environmental Factors Index

Now that we evaluated the costs and discovered their correlation with the porting factors we will conduct a discussion on the results of our evaluation.

Chapter 6

Discussions on Porting Costs

In this chapter we discuss conclusions of our porting work, including the limitations of the porting model we used and factors for these limitations, namely the dependency between porting tasks. We also compare our results with the results of the first paper that published a porting model to understand what were the differences between our and their results and conclusions. Finally, we also present general difficulties that we extracted from our porting.

6.1 Conclusions of our porting

In this section we present observations about the porting process and the porting costs, focusing on the limitations of the porting model we used and on a comparison between our results and the results of Tanaka et al.

6.1.1 Limitations of the porting model

The model assumes that the tasks are executed in sequential order, which is not true. The tasks are rather executed in a non-linear fashion. For example while building for the target environment, installing the binaries in this environment and testing the ported application, we also had discussions about the difficulties and errors we encountered so that we could later come back and solve the inconsistencies. However there was no easy way to describe this dynamic so we chose to represent the task as they would come one after another. The non-linearity of this porting model causes problems while computing the porting costs. If one wants to compute the costs with minimum error it means that more time must be allocated to the *Progress tracking* subtask. This strategy might not bring the best results if the fixed total time of porting is computed in advance because allocating more time for progress tracking means that other porting tasks must have lesser time allocated. A good strategy here would compute first the accepted error of man-hours present in the final porting costs so that an acceptable ammount of time is allocated for tracking the progress of the project. For our

porting process it was not vital to have a small error in the porting costs, we were interested to see an approximative distribution of time per tasks so that we could answer questions as: where did we spend most time and why? or what was the relation between development and testing? For this we allocated an hour each week for tracking the status of the project, the planned objectives for the future and the major difficulties we met in the respective week. It is unclear, however, if more time spent on progress tracking would improve the quality of other tasks. Although we can only guess, more time spent on tracking the working hours would mean that the developer has a better sense of the time spent on each task, thus better managing multiple tasks at a time in the future.

As discussed above, we had errors while extracting the porting costs for each subtask in Table 1. There are two reasons for this issue: the progress tracking format was not descriptive enough and the model we used for extracting the porting costs had shortcomings. Firstly, the progress tracking format should have an intrinsic expresiveness that would make it easy for the user to express the non-linearity of the porting tasks. To create this kind of format we need to first find a way to describe the non-linearity of the porting process, which is not a trivial task whatsoever. Secondly, we expect a conversion error from the classical progress tracking format (i.e., list of bullets for current status and planning) to the porting model presented in section 2. This happens because there is no straightforward way of converting a bullet as "IM is booting, having config issues" (this example is taken from our progress tracking) into clear and independent porting tasks. This is subject to interpretation one might say that this bullet might be broken down into the following porting tasks:

- Reviewing inconsistencies between source and remote environments
- Testing in target environment
- Discussions
- Surveying target OS

while another would break the bullet down into the following porting tasks:

- Reviewing inconsistencies between source and remote environments
- Testing in target environment

It very much depends on the human factors and how the developers decide to work with the provided tools and systems. Furthermore, to have a more accurate conversion there must be time considerents: how much time did we spent on this bullet? and how do we divide this time between porting tasks?

6.1.2 Dependencies between porting tasks

The next discussion point focuses on the porting tasks dependency graph. It would be very difficult to sketch a complete graph between all tasks. Instead, we can focus on specific

areas of the graph that are somehow easier to understand and sketch. The first area is the dependency between *Testing* and all subtasks in *Building for target environment*. There is a continous feedback between the testing phase and the development phase. While porting, we built binaries, installed them on target environment, tested them and expected to see errors of linking, problems with the installation or actual inconsistencies and problems with dependencies that would be solved in a future iteration of this process. This iterative process with continous feedback between testing and development is a building block of the Agile methodology. If we used other software methodology as Waterfall maybe the dependency would be completely different. The second area we focus on is the dependency between *General duties* and all other subtasks. Documentation is linked to all subtasks because we need to include all relevant information in the documents we produce, progress tracking must reflect work executed in each subtask and discussions are by design focused on trying to understand each part of the porting process. That being said, it is hard to imagine how *General duties* would not be a substantial part of the porting process. It would be interesting to see if a modification in the allocated time for a subtask such as *Discussions* would reduce or increase the needed time for other subtasks, otherwise if other tasks are not affected by this modification, there should be no interest in allocating more time for discussing.

6.2 Comparison between our results and the results of Tanaka et al.

At this moment we discussed the limitations and open problems that appear in the model of porting we used. Next, we make a comparison between our results for porting costs and the results that appeared in the first paper that presented a basis for our porting model [16]. The results can be found in Table 6 of the paper we referenced. We will compare the subtotals per tasks and discuss the reasons of the differences between the two. In terms of *Advance preparations* we spend a total of 14.35% of our total time while Tanaka et al. spend 33.4%. There are two major reasons for reduced time in our case. First, we were interested in delivering builds for the target environment as fast as possible, sacrificing the time spent on understanding each part of the system we were trying to port. We found no easy way to understand the components of the system by surveying the documentation and the program for porting so we decided to offload this work on the build system (i.e., when the compiler threw errors for a component, we went to that component, understood it and solved the problems). Second, we worked with the Agile methodology in mind, combining subtasks from different tasks in the same iteration. For this reason we lost the focus on the *Advance preparations* because we were also interested in subtasks from *Building for target environment*, for example. Tanaka et al. seem to work with a Waterfall methodology. This might explain the increased time in this initial stage.

Next, we compare the *Building for target environment* in our model with *Target testing* from Tanaka et al. Although there are significant differences in the subtasks from the two models,

we are interested in a comparison of development time (i.e., actual work focused on solving errors and inconsistencies, building binaries, etc.). It seems that Tanaka et al. put the time spent on solving errors and the time spent on testing in the same category. This makes the comparison difficult as we do not know how to divide this time. However, our time spent in this stage is 32.71% while Tanaka et al. spend 27.8%. The most time consuming task for us was to work with the build system, which is not the case for them. This shows the major difference between our project and theirs. While our porting was focused on extracting components from a larger ecosystem into a target environment, their project was focused on porting a whole application from one environment to another.

For *Testing* we assume that they spent *Linked test on target*$/2+$*Workstation testing* $= 10.5+11.4 = 21.9\%$. We divided the linked test on target because we assumed that half of this time was allocated to development and half of it was allocated to testing. Their time for *General duties* is of 27.4%. These two times are very similar with our project (i.e, 24.53% and 28.26%). This might mean that testing and general duties represent a major part in each software porting project, thus when evaluating the costs in advance, special attention must be payed to these tasks.

6.3 General difficulties

This section presents the general impediments we faced during our porting. The difficulties were extracted from the particular technical difficulties we faced during the porting process.

Lacking documentation. Lacking written documentation about how the system works means that the developer must either figure out the system alone or must communicate with other developers in order to gather information about the system. This adds overhead to the porting process as documentation through communication is slower than documentation through written text.

Inconsistencies between environments. This difficulty corresponds to the degree of which the application is portable [13] between two given environments. If the degree of portability is too low (this depends entirely on the application), then the developer will be faced with many inconsistencies between the source and target environments that will be reflected in the cost of porting (i.e., man-hours).

Use of tools. Using inadequate development and testing tools introduces additional overhead in the porting process. This happens when there is a mismatch between the version of a tool available in the development environment and the version expected by the project. This introduces additional overhead in installing the proper tools. Moreover if the correct version of the tool cannot be found anymore, more overhead is added by finding workarounds.

Understanding the system. Software complexity is a multi-dimensional problem, it includes: structural, computational, logical, conceptual and textual complexity [6]. There is no easy

way to understand the system, so the developer will be faced with the task of understanding the architecture diagrams, huge code base, written documents and tutorials either when the porting starts or during the porting process.

Chapter 7

Conclusions

We succeeded in porting the IxOS infrastructure on ARM boards. We separated the relevant components for our porting (i.e., InterfaceManager and IxStack) from IxOS so that they can be run on any ARM-based Linux distribution. At the moment of writing the project is in the proof-of-concept stage. If there is interest for continuing the project or integrating it in other projects inside the company, we provided the necessary environment for deploying it.

We have extracted the porting costs for porting IxOS infrastructure on ARM boards. Thus we understood what the weaknesses and the strengths of our project were. The lack of understanding of the project structure and project use cases proved itself to be an important factor during the process of porting. Because of this reason we had to spend additional time on testing the system and understanding its components. This time might have been better allocated on solving problems and inconsistencies. A strength of our project was the fact that the the ported program had a high degree of portability. This helped us to shrink the volume of inconsistencies between the source environment and the target environment.

We succeeded in creating a more accurate and comprehensive software porting model with regards to today's standards of software engineering starting from the model presented in [8,16] and from our project specific needs. We contributed to this model by making it more generic and allowing other software porting projects to easily map their needs on this model.

Finally, we have provided a discussion on the limitations of the model we created and provided a comparison between our porting results and the results presented in the first work [8] that came up with the model we used as a basis for our generic porting model.

Chapter 8

Further Work

We aim to make the testing infrastructure portable to as many environments (OS, compiler, architecture) as possible. This would be a big win for the software project because people interested in using the portable software parts as drop-in components on any Linux-based system could do it with little effort, supposing that a C++ compiler would exists for the specific architecturure the software will be installed on.

We plan to explore other aspects of our port to ARM, as system performance of while running network testing suites. To achieve this goal we should compare our solution in the target environment with the same solution in the source environment. We expect to see better results in the new environment for some network testing suites than in the old environment. Furthermore, we plan to compare our solution with other open-source network testing tools.

To complete the analysis of the factors that affected the porting costs we plan to analyze the characteristics of the program to be ported. We want to analyze the program size and contents and the content of the changes needed for porting. By doing this we aim to find a direct correspondence between the program to be ported and specific porting subtasks (e.g., *Solving inconsistencies between source and target environment*).

Finally, we want to analyze the porting improvements guidelines presented in [8] and map them on our porting process. However, we do not plan to restart the porting process while mapping these guidelines, instead we want to have a discussion and draw conclusions based on them.

Bibliography

[1] Cross-porting software. `https://wiki.osdev.org/Cross-Porting_Software`, Sept 2019.

[2] DE Bodenstab, Thomas F Houghton, Keith A Kelleman, George Ronkin, and Edward P Schan. The UNIX system: UNIX operating system porting experiences. *AT&T Bell Laboratories Technical Journal*, 63(8):1769–1790, 1984.

[3] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approaches—a survey. *Annals of software engineering*, 10(1):177–205, 2000.

[4] B. R. Callahan. I ported the new hare compiler to OpenBSD. =https://briancallahan.net/blog/20220427.html, April 2022.

[5] Luiz Fernando Capretz. Bringing the human factor to software engineering. *IEEE software*, 31(2):104–104, 2014.

[6] Lem O Ejiogu. A simple measure of software complexity. *ACM SIGPLAN Notices*, 20(3):16–31, 1985.

[7] William B Frakes and Christopher J Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–ff, 1995.

[8] Mitsuari Hakuta and Masato Ohminami. A study of software portability evaluation. *Journal of Systems and Software*, 38(2):145–154, 1997.

[9] Steven C Johnson and Dennis M Ritchie. UNIX time-sharing system: Portability of c programs and the UNIX system. *The Bell System Technical Journal*, 57(6):2021–2048, 1978.

[10] William Frederick Jolitz and Lynne Greer Jolitz. Porting UNIX to the 386: A practical approach. *Dr. Dobb's Journal*, 16(1):16–46, 1990.

[11] A Kanai, T Furuyama, and M Takahashi. A cost model for software conversion based on program characteristics and a converter effect. In *1992 Proceedings. The Sixteenth Annual International Computer Software and Applications Conference*, pages 63–64. IEEE Computer Society, 1992.

[12] James D. Mooney. Strategies for supporting application portability. *Computer*, 23(11):59–70, 1990.

[13] James D Mooney. Developing portable software. In *Information Technology*, pages 55–84. Springer, 2004.

[14] Malcolm J Morgan. Controlling software development costs. *Industrial Management & Data Systems*, 1994.

[15] Joël Porquet. Porting Linux to a new processor architecture, part 1: The basics. `https://lwn.net/Articles/654783/`, Aug 2015.

[16] Toshikiyo Tanaka, M Hakuta, N Iwata, and M Ohminami. Approaches to making software porting more productive. In *Proceedings of the 12th TRON Project international Symposium*, pages 73–85. IEEE, 1995.

[17] Andrew S Tanenbaum, Paul Klint, and Wim Bohm. Guidelines for software portability. *Software: Practice and Experience*, 8(6):681–698, 1978.

[18] Stephen R Walli. The posix family of standards. *StandardView*, 3(1):11–17, 1995.

[19] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.