The Transition from Single-Platform to Multi-Platform Software Development

Juris Reinfelds

juris@nmsu.edu

Klipsch School of Electrical and Computer Engineering

New Mexico State University, Box 30001 Dept 3-0, Las Cruces NM 88003 USA

Abstract

In this paper we will observe that there were sound reasons why multi-platform software development was once regarded as impossible. When the reasons disappeared, the belief of impossibility remained. The first ports of UNIX helped to destroy that belief, opening a floodgate to subsequent UNIX ports and multi-platform software developments.

Introduction

Like the four-minute mile, a multi-platform operating system was unthinkable until it was achieved and then it became commonplace. Subsequent generations find it difficult to understand why it was once regarded as an unsurmountable problem.

Let us examine the reasons why multi-platform operating systems were not feasible thirty years ago and how these reasons had largely disappeared about twenty years ago. For quite some time hardly anyone noticed that these reasons had disappeared. The tradition of single-platform development and the belief that ported software cannot be efficient were just too strong.

It is interesting to note that even today portability mechanisms have not attracted much interest from academic computer scientists. Operating system and software engineering books do not have chapters on portability methods. The word *"portability"* seldom appears in the index of those books. Professional journals carry very few, if any, articles on how to achieve efficient and elegant portability. Yes, there are articles on abstract machines for which we write code that is portable to all computers on which the abstract machine is implemented. Some of these abstract machines are elegant, but there is little effort to relate these abstract machines to abstract machines representing computer hardware so as to achieve an elegant and efficient solution. In an academic vacuum, the remarkable success of multi-platform software development has been achieved by practitioners *"in the trenches,"* using *"proof by doing"* methods, quite often at a high cost to elegance and simplicity.

In science it is often the case that when the time of a new idea "has come," several scientists, or teams of scientists, start to develop that idea independently and achieve a result more or less at the same time. This was the case with the first ports of UNIX. The Bell Labs [7] team and the Wollongong [9] team independently chose similar computers (Interdata 8/32 for Bell and Interdata 7/32 for Wollongong), performed the port and surprised each other

with a *"verification",* by what turned out to be entirely different implementation methods, of the other's *"proof by doing."* By historical accident rather than design, not only the implementation methods, but also the side effects of the two ports were complementary.

While doing the Bell port, Ritchie and Johnson [7] developed and improved many portability tools, as described in more detail in Steve Johnson's paper in this booklet. At Wollongong, Richard Miller [8] showed that a well designed operating system kernel can be made to run on another computer architecture, at least as efficiently as the assembler-coded operating system that was tailor-made for that architecture by its manufacturer. He also "showed by doing" that all this can be done in a reasonable time period and without the need for consultations with the kernel authors or the machine architecture designers. Together these results opened the subsequent floodgate of UNIX ports and multi-platform software development a bit wider than they would have otherwise been at that time.

Diverse Computer Architecture Era

In the early days of computers computer designers worshiped diversity. In their excellent book on computer structures, Gordon Bell and Allen Newell [3] describe about 250 of the estimated 1,000 computer architectures implemented up to 1970 when their book was written. Almost all of these computers were word-addressed. Word sizes were diverse, ranging from 8 bits through 12, 16, 18, 24, 32, 36, 48, 60 to 64 bits per word. Characters were represented by 6-bit bytes in some computers as well as by 8-bit bytes in others. Only the 48-bit word size could accommodate either representation without spare bits. Since memories were small and expensive, all kinds of clever tricks, unique to a specific architecture, were introduced to make use of the spare bits and nibbles. In the CPU there were data registers, index registers and other special purpose registers that were unique to each family of computers. I/O required very specialized I/O channel processors and therefore I/O handling was also unique to each family of computers. This caused Stone and Sieworek [4] to remark (p. 205):

> *"Input/Output programs are among the most difficult to write for a number of reasons...."*

All this subtly changed with the introduction of the PDP-11 computer by Digital Equipment Corporation in 1969. The PDP-11 had byte addressable memory, general-purpose registers and memory-mapped I/O without special I/O processors. This removed the major obstacles to efficient porting of system programs, but no one noticed until the first ports of UNIX succeeded some eight years later.

The IBM-360 family of computers had already introduced byte addressed memory and general purpose registers in 1964 but their I/O handling involved channel processors. Several computer companies (such as Interdata) subsequently introduced very similar architectures to the IBM-360 and simplified I/O handling considerably. Hence, by 1979 when the PDP-11 appeared, there were a number of computer architectures between which porting was feasible and relatively straightforward, but as we said before, no one noticed...

The Awakening of Multi-Platform Software Development

The feasibility for porting was not a property that users or researchers were concerned with when designing computer architectures. In fact, practical portability

was ignored and if it was studied, the framework of the problems that were studied were quite unrealistic. For example, in the seventies there were several attempts to develop tools and methods for software porting from every architecture to any architecture whatsoever. One approach, called Janus, was to port at assembler level via a lowest-common-denominator machine, which in this case was a basic one-register machine. As a consequence, the ported code lost a lot in efficiency and elegance. There were too many computer architecture dependencies at the assembler level and this method did not succeed in producing a commercially viable system.

Another approach, (Solo Operating System by Per Brinch Hansen [2]), was to write the code in Concurrent Pascal and to rely on the portability of the Pascal language. Unfortunately, the Concurrent Pascal dialect was not popular and the "Pascal machine" was too far removed from the abstract machines representing computer architectures. In addition, the language was too restrictive for the needs of system programming, so that this line of approach did not produce a commercially viable system either.

Operating system designs in England had for a long time been expressed in terms of abstract machines, (Ibbett [1]), but since the MU-5 architecture was commercially exploited by one computer company, they were not interested in operating system portability to other architectures. By contrast, the C programming language was invented to express the mechanisms of an operating system with just the right amount of abstraction to encourage elegant code without sacrificing of efficiency of code execution. Initially, C and its compiler were designed primarily for just one architecture, the PDP-11, but with the experience gained from the first ports, C evolved into an effective, efficient and elegant portability tool.

Thus, by 1979 all essential elements for successful porting of system software were in place. There was a growing pool of computer architectures with byte-addressable memories, with general purpose registers and with memory mapped I/O that did not use special I/O channel processors. The C language encouraged elegant programming without sacrificing efficiency of execution. There was an excellent, portable C compiler for which one could write code generators for a new computer architecture with relative ease. Only the fear of the unknown remained.

When poorly designed single platform programs were ported to other platforms, the ports were often inefficient or failed altogether. This reinforced the old belief that multi-platform code must be inefficient. When programs were designed for multi-platform implementation from the start, the results were much better. By the end of the eighties, multi-platform development finally became the norm rather the exception.

Conclusions

The successful porting of UNIX to most of today's computer architectures convinced the skeptics and accelerated the development of multi-platform software systems. A number of useful lessons were learned in the transition period.

(i) Efficient portability is enabled by a small number of important architectural commonalities. For successful porting of operating systems and for many other multi-platform software systems only three key features are absolutely essential:

a) byte-addressable memory

b) general-purpose registers

c) memory-mapped I/O without special processors.

(ii) A successful implementation language of multi-platform systems has to balance the level of abstraction (used by programmers to enhance simplicity and elegance of programs) against the efficiency of translation of these abstractions into the model of computation of a computer architecture so as to permit an efficient execution of code).

(iii) A poorly designed software system cannot be improved by porting. It can only become worse, because porting tends to show up hidden defects.

Today, we look at parallel computers and the writing of programs for parallel computers in much the same way as we looked at the portability of programs for the diverse architectures of the mainframes of the fifties and sixties. There are too many different parallel architectures. There are too many different models of computation for parallel programs. A program that runs well on one architecture, typically runs poorly or not at all on others. Large amounts of effort and resources are expended in the design and implementation of software systems that will translate programs written for one architecture into programs that should run well on another. So far, success has been limited and our previously concluded lessons (i), (ii), and (iii) suggest good reasons why.

The success of porting of sequential programs was limited until computer architectures standardized a small number of key features. Let us hope that something similar happens to parallel architectures soon. The difficulty is that it is difficult to predict *a priori,* as to which features are the key features that must be standardized. If we can learn from history that it is advantageous to know what are the essential commonalities of parallel architectures, then we will not have to wait until someone stumbles upon them accidentally.

References

[1] R. N. Ibbett, *"The MU-5 Computer System,"* Springer Verlag, (1979).

[2] P. Brinch Hansen, *"The Solo Operating System,"* Software Practice and Experience, Vol. 6, (1976).

[3] C. Gordon Bell, Allen Newell, *"Computer Structures,"* McGraw-Hill, (1971).

[4] Harold S. Stone, Daniel P. Siewiorek, "Introduction to Computer Organization and Data Structures: PDP-11 Edition," McGraw-Hill, (1975).

[5] Andrew S. Tanenbaum, *"Structures Computer Organization,"* Third Edition, Prentice-Hall, (1990).

[6] William Stallings, *"Computer Organization and Architecture,"* Fourth Edition, Prentice-Hall, (1996).

[7] S. C. Johnson, D. M. Ritchie, *"Portability of C Programs and the Unix System"*, Bell Sys. Tech. Journal, 57, 6, p. 2021-2048, (1978).

[8] Richard Miller, *"Unix, a Portable Operating System?"*, ACM Op. Sys. Review, 12, p. 32-37, (1978).

[9] Juris Reinfelds, "The First Port of Unix", Annals of Hist of Computing, 11, 3, p. 208-210, (1989).