Porting

In software engineering, **porting** is the process of adapting software for the purpose of achieving some form of execution in a computing environment that is different from the one that a given program (meant for such execution) was originally designed for (e.g., different CPU, operating system, or third party library). The term is also used when software/hardware is changed to make them usable in different environments.[1][2]

Software is *portable* when the cost of porting it to a new platform is significantly less than the cost of writing it from scratch. The lower the cost of porting software relative to its implementation cost, the more portable it is said to be.

Contents

Etymology

The term "port" is derived from the Latin *portāre*, meaning "to carry".[3] When code is not compatible with a particular operating system or architecture, the code must be "carried" to the new system.

The term is not generally applied to the process of adapting software to run with less memory on the same CPU and operating system, nor is it applied to the rewriting of source code in a different language (i.e. language conversion or translation).

Software developers often claim that the software they write is *portable*, meaning that little effort is needed to adapt it to a new environment. The amount of effort actually needed depends on several factors, including the extent to which the original environment (the *source platform*) differs from the new environment (the *target platform*), the experience of the original authors in knowing which programming language constructs and third party library calls are unlikely to be portable, and the amount of effort invested by the original authors in only using portable constructs (platform specific constructs often provide a cheaper solution).

History

The number of significantly different CPUs and operating systems used on the desktop today is much smaller than in the past. The dominance of the x86 architecture means that most desktop software is never ported to a different CPU. In that same market, the choice of operating systems has effectively been reduced to three: Microsoft Windows, macOS, and Linux. However, in the embedded systems and mobile markets, portability remains a significant issue, with the ARM being a widely used alternative.

International standards, such as those promulgated by the ISO, greatly facilitate porting by specifying details of the computing environment in a way that helps reduce differences between different standards-conforming platforms. Writing software that stays within the bounds specified by these standards represents a practical although nontrivial effort. Porting such a program between two standards-compliant platforms (such as POSIX.1) can be just a matter of loading the source code and recompiling it on the new platform. However, practitioners often find that various minor corrections are required, due to subtle platform differences. Most standards suffer from "gray areas" where differences in interpretation of standards lead to small variations from platform to platform.

There also exists an ever-increasing number of tools to facilitate porting, such as the GNU Compiler Collection, which provides consistent programming languages on different platforms, and Autotools, which automates the detection of minor variations in the environment and adapts the software accordingly before compilation.

The compilers for some high-level programming languages (e.g. Eiffel, Esterel) gain portability by outputting source code in another high level intermediate language (such as C) for which compilers for many platforms are generally available.

Two activities related to (but distinct from) porting are emulating and cross-compiling.

Porting compilers

Instead of translating directly into machine code, modern compilers translate to a machine independent intermediate code in order to enhance portability of the compiler and minimize design efforts. The intermediate language defines a *virtual machine* that can execute all programs written in the intermediate language (a machine is defined by its language and vice versa).[4] The intermediate code instructions are translated into equivalent machine code sequences by a *code generator* to create executable code. It is also possible to skip the generation of machine code by actually implementing an interpreter or JIT for the virtual machine.[5]

The use of intermediate code enhances portability of the compiler, because only the machine dependent code (the interpreter or the code generator) of the compiler itself needs to be ported to the target machine. The remainder of the compiler can be imported as intermediate code and then further processed by the ported code generator or interpreter, thus producing the compiler software or directly executing the intermediate code on the interpreter. The machine independent part can be developed and tested on another machine (the *host machine*). This greatly reduces design efforts, because the machine independent part needs to be developed only once to create portable intermediate code.[6]

An interpreter is less complex and therefore easier to port than a code generator, because it is not able to do code optimizations due to its limited view of the program code (it only sees one instruction at a time, and you need a sequence to do optimization). Some interpreters are extremely easy to port, because they only make minimal assumptions about the instruction set of the underlying hardware. As a result, the virtual machine is even simpler than the target CPU.[7]

Writing the compiler sources entirely in the programming language the compiler is supposed to translate, makes the following approach, better known as *compiler bootstrapping*, feasible on the target machine:

1. Port the interpreter. This needs to be coded in assembly code, using an already present assembler on the target.
2. Adapt the source of the code generator to the new machine.
3. Execute the adapted source using the interpreter with the code generator source as input. This will generate the machine code for the code generator.

The difficult part of coding the optimization routines is done using the high-level language instead of the assembly language of the target.

According to the designers of the BCPL language, interpreted code (in the BCPL case) is more compact than machine code; typically by a factor of two to one. Interpreted code however runs about ten times slower than compiled code on the same machine.[8]

The designers of the Java programming language try to take advantage of the compactness of interpreted code, because a Java program may need to be transmitted over the Internet before execution can start on the target's Java Virtual Machine.

Porting of video games

Porting is also the term used when a video game designed to run on one platform, be it an arcade, video game console, or personal computer, is converted to run on a different platform. From the beginning of video games through to the 1990s, "ports", at the time often known as "conversions", were often not true ports, but rather reworked versions of the games due to limitations of different systems. For example, the 1982 game *The Hobbit*, a text adventure augmented with graphic images, has significantly different graphic styles across the range of personal computers that its ports were developed for.[9] However, many 21st century video games are developed using software (often in C++) that can output code for one or more consoles as well as for a PC without the need for actual porting (instead relying on the common porting of individual component libraries).[9]

Porting arcade games to home systems with inferior hardware was difficult. The ported version of Pac-man for the Atari 2600 omitted many of the visual features of the original game to compensate for the lack of ROM space and the hardware struggled when multiple ghosts appeared on the screen creating a flickering effect. The poor performance of the ported Pac-Man is cited by some scholars as a cause of the video game crash of 1983.[10]

Many early ports suffered significant gameplay quality issues because computers greatly differed.[11] Richard Garriott stated in 1984 at Origins Game Fair that Origin Systems developed computer games for the Apple II series first then ported them to Commodore 64 and Atari 8-bit, because the latter machines' sprites and other sophisticated features made porting from them to Apple "far more difficult, perhaps even impossible".[12] Reviews complained of ports that suffered from "Apple conversionitis",[13] retaining the Apple's "lousy sound and black-white-green-purple graphics";[14][15] after Garriott's statement, when Dan Bunten asked "Atari and Commodore people in the audience, are you happy with the Apple rewrites?" the audience shouted "No!" Garriott responded, "[otherwise] the Apple version will never get done. From a publisher's point of view that's not money wise".[12]

Others worked differently. Ozark Softscape, for example, wrote *M.U.L.E.* for the Atari first because it preferred to develop for the most advanced computers, removing or altering features as necessary during porting. Such a policy was not always feasible; Bunten stated that "M.U.L.E. can't be done for an Apple",[11] and that the non-Atari versions of *The Seven Cities of Gold* were inferior.[16] *Compute!'s Gazette* wrote in 1986 that when porting from Atari to Commodore the original was usually superior. The latter's games' quality improved when developers began creating new software for it in late 1983, the magazine stated.[17]

In porting arcade games, the terms "arcade perfect" or "arcade accurate" were often used to describe how closely the gameplay, graphics, and other assets on the ported version matched the arcade version. Many arcade ports in the early 1980s were far from arcade perfect as home consoles and computers lacked the sophisticated hardware in arcade games, but games could still approximate the gameplay. Notably, *Space Invaders* on the Atari VCS became the console's killer app despite its differences,[18] while the later *Pac-Man* port was notorious for its deviations from the arcade version.[19] Arcade-accurate games became more

prevalent starting in the 1990s, starting with the Neo Geo system from SNK, which offered both a home console and an arcade system with more advanced versions of the home console's main hardware. This allowed near-arcade perfect games to be played at home. Further consoles such as the PlayStation and Dreamcast, also based on arcade hardware, made arcade-perfect games a reality.[9]

"(Console) port" is a game that was originally made for a console (such as Wii or Xbox 360) before an identical version is created which can be played on a personal computer or any other console. This term has been widely used by the gaming community. The process of porting a game from a console to a PC is often regarded negatively due to the higher levels of performance that computers generally have being underutilized, partially due to console hardware being fixed throughout their run (with games being developed for console specs), while PCs become more powerful as hardware evolves, but also due to ported games sometimes being poorly optimized for PCs, or lazily ported. Whilst broadly similar, architectural differences may exist such as the use of unified memory on a console.

See also

- Software portability
- Language binding
- Console emulator
- List of system quality attributes
- Source port
- Write once, compile anywhere
- Poshlib
- Cross-platform
- Program transformation
- Meaning of *unported*

Notes

1. Whitten, D.E.; Demaine, P.A.D. (March 1975). "A machine and configuration independent Fortran: Portable Fortran". *IEEE Transactions on Software Engineering*. **SE-1** (1): 111–124. doi:10.1109/TSE.1975.6312825 (https://doi.org/10.1109%2FTSE.1975.6312825). S2CID 16485156 (https://api.semanticscholar.org/CorpusID:16485156).
2. "Portability Issues" (https://www.gnu.org/software/sather/docs-1.2/tutorial/fortran-portability.html). ".. discusses .. portability of .. Fortran"
3. "port, v.2" (http://www.oed.com/view/Entry/148098). *Oxford English Dictionary (OED Online)*. Oxford University Press. Retrieved December 21, 2017. "Origin: Of multiple origins. Partly a borrowing from French. Partly a borrowing from Latin. Etymons: French *porter*; Latin *portāre*. ... 1. *trans.* To carry, bear, or convey; to bring."
4. Tanenbaum 1984, p. 3. §1.1 Languages,Levels, and Virtual Machines describes the terms and their relations.
5. Tanenbaum 1984, p. 2. Ch. 1 Introduction explains translation and interpretation.
6. Richards & Whitby-Strevens 1984, p. 124. §7.1 Introduction explains compiler portability using intermediate code.
7. Richards & Whitby-Strevens 1984, p. 133. §7.4 The bootstrapping process and INTCODE explains the role of the INTCODE interpreter.
8. Richards & Whitby-Strevens 1984, p. 136. §7.4.3 Example gives an example translation of a BCPL program into INTCODE for the interpreter.

9. Grabarczyk, Pawel; Aarseth, Espen (2019). "Port or conversion? An ontological framework for classifying game versions". Unknown parameter |conference= ignored (help);
10. Nicoll, Benjamin (2015). "Bridging the Gap: The Neo Geo, the Media Imaginary, and the Domestication of Arcade Games". *Games and Culture*. doi:10.1177/1555412015590048 (https://doi.org/10.1177%2F1555412015590048). S2CID 147981978 (https://api.semanticscholar.org/CorpusID:147981978).
11. Bunten, Dan (December 1984). "Dispatches / Insights From the Strategy Game Design Front" (http://www.cgwmuseum.org/galleries/index.php?year=1984&pub=2&id=19). *Computer Gaming World*. p. 40. Retrieved 31 October 2013.
12. "The CGW Computer Game Conference" (http://www.cgwmuseum.org/galleries/index.php?year=1984&pub=2&id=18). *Computer Gaming World* (panel discussion). October 1984. p. 30. Retrieved 31 October 2013.
13. Dunnington, Benn; Brown, Mark R.; Malcolm, Tom (January–February 1987). "64/128 Gallery" (https://archive.org/stream/info-magazine-13/Info_Issue_13_1987_Jan-Feb#page/n13/mode/2up). *Info*. pp. 14–21.
14. Stanton, Jeffrey; Wells, Robert P.; Rochowansky, Sandra; Mellid, Michael, eds. (1984). *The Addison-Wesley Book of Atari Software* (https://archive.org/stream/Atari_Software_1984#page/n21/mode/2up). Addison-Wesley. pp. 12, 21, 44, 126. ISBN 0-201-16454-X.
15. Bernstein, Harvey (May 1985). "Beyond Castle Wolfenstein" (https://archive.org/stream/1985-05-anticmagazine/Antic_Vol_4-01_1985-05_New_Super_Ataris#page/n81/mode/2up). *Antic*. p. 83. Retrieved 8 January 2015.
16. Bunten, Dan. "The Game Collection" (http://www.ozarksoftscape.com/the-game-collection.html). *Ozark Softscape M.U.L.E*. Retrieved 2017-10-04.
17. Yakal, Kathy (June 1986). "The Evolution of Commodore Graphics" (https://archive.org/details/1986-06-computegazette/page/n35). *Compute!'s Gazette*. pp. 34–42. Retrieved 2019-06-18.
18. Kent, Steven (2001). *Ultimate History of Video Games*. Three Rivers Press. p. 190. ISBN 0-7615-3643-4.
19. Kent, Steven (2001). "The Fall". *The Ultimate History of Video Games*. Three Rivers Press. pp. 237–239. ISBN 978-0-7615-3643-7.

References

- Richards, Martin; Whitby-Strevens, Colin (1984). *BCPL, the language and its compiler*. ISBN 0-521-28681-6.
- Tanenbaum, Andrew S. (1984). *Structured computer organization*. ISBN 0-13-854605-3.