Speeding up Code-Compilation for Botball

Philias Kuntsche*, Sebastian Munkhbat,
Department of Computer Science
Höhere Technische Bundes-Lehr- und Versuchsanstalt Wiener Neustadt
(Federal Technical Secondary College)
Wiener Neustadt, Austria
*Corresponding author email: kuntsche.philias@student.htlwrn.ac.at

***Abstract*—To compile code as efficiently as possible and thus work much faster is a goal of most programmers, particularly in the context of the Botball competition. The authors present two concepts to increase the speed of compilation, cross- and distributed compiling. They describe in detail how the concepts work theoretically and how the two concepts differ. The authors also discuss the practical implementation and the problems or approaches there are and how they can be solved. Finally, further ideas for making compilation faster are described. The impact of this work extends beyond the Botball competition, informing roboticists and engineers about the problems on how to increase the efficiency of compiling and programming code and what problems there are. To help many developers, all resources are online**

***Index Terms*—distributed compilation, virtual images, docker, QEMU**

I. Introduction

Compilation is a significant factor for large projects, such as the Botball competition, optimizing process management and compilation is crucial for maximizing effectiveness, particularly the efficiency. The primary concern is the performance of the robot, in the case of the authors the Wombat [4] used in the competition, which, due to its small form factor, has inferior performance. This work aims to improve the speed of compiling code and proposes two approaches: cross- and distributed compilation. In this case distributed compilation means to send code to a virtual machine on a server. This machine compiles the code and sends it to the Wombat. Cross-compilation involves compiling code for a targeted system on a more powerful computer, which may have a different operating system and processor architecture.
On the other hand, distributed compilation uses a third-party system, such as a server, to compile the code, send the binary to and have it executed on the target. The idea is to run a virtual machine on a powerful server, which compiles the code for the Raspberry Pi [3], which is used in the Wombat Controller, directly on the desired architecture and thus speeding up the compilation.
This study highlights the issues with strategies aimed at improving compilation efficiency and proposes solutions to address them.

II. Study of Literature

As stated by the authors of "Cross-Compiling tutorial with Scratchbox", cross-compilation was developed late, code was written for a specific machine that had a specific purpose [1]. But after realising that some processors have more computing power than others and that the code is used in many machines and for various purposes, it was easy to see that in many cases compiling the application on a different machine would take less time than compiling it on a machine where it is to be used, and so the idea of cross-compilation was born. Therefore, the use of cross-compilation only makes sense if it reduces the time required for compiling. To date, cross-compilation has been implemented and documented and so far there are plenty of works or studies that deal with the problems and implementation. This study focuses on the detailed implementation, problems that occurred, suggestions for solutions and how it can be used in the Botball competition. Cross-compilation can in most cases be somewhat complex and fragile [2]. The reasons for this are that there are many different methods of implementing cross-compilation and all of them have their own challenges. Another problem is how the libraries used are structured. It can also be difficult to ensure compatibility between the host and target systems.

III. Methods

This section introduces and explains how the authors attempted to set up cross- and distributed compilation. Important information about the tools used is described further below.

A. Programming Language

The programming language used in this work is C++. As seen in Table I the authors use C++20. The newest option is C++23, this work opts to use C++20 as it is the current standard used by most programmers. It has various advantages and new features opposed to the previous versions. One such feature is the three way comparisons operator, which could be used to measure the distance between the target and the robot and take the appropriate action, or to tell what color a certain object is on a camera or sensor.

B. QEMU

QEMU, short for Quick Emulator, is a free and open-source emulator that uses dynamic binary translation to emulate a foreign architecture. It provides a range of hardware and device models, allowing it to run various guest operating

Usage	Tool	Version/Specification
Programming Language	C++	C++20
Virtualization Software	QEMU	8.2.0
	Docker	23.0.3
Hardware	Botball Kit	2023
Operating System	Wombat OS	ver. 30.2.4

TABLE I: An overview of tools used for the study

systems. QEMU can also work with Kernel-based Virtual Machines (KVM) to achieve near-native virtual machine speeds for same-architecture guests. QEMU also allows for execution of processes on a user-level even on different architectures. It allows applications, compiled for one architecture, to operate on another [9]. The version and specifications can be seen in Table I.

C. Wombat

The KIPR Wombat is the current robot controller used in Botball. It has been selected for all research as the computer that will receive the final compiled binary file and execute it, as it can be easily be mounted onto a robot as well as connect to it to build programs. It has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU and one gigabyte of RAM. Additionally the wombat has several plugs, which makes connecting servos and motors easy, as well as allowing to test them using a menu option on the Wombat operating system. The network card of the Wombat allows it to host or connect to a network, enabling communication between the Wombat and other computers.

D. Docker

Docker is an open-source platform that allows the user to automate the deployment, scaling, and management of applications within containers. It has been selected to compile the code inside a container to provide the necessary isolation and reproducibility needed by such a project. These containers are lightweight and standalone packages that contain everything needed to run an application, including the code, runtime, system tools, libraries, and settings. They are isolated from each other and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels [10].

IV. Cross Compilation in Theory

The computer used in this study, or the compiling computer, is a modern laptop running the Windows operating system. Its processor is the Intel Core i5 12500H [5].

The task involves compiling code for the Raspberry Pi and sending it via an SSH connection within the same network. This approach is expected to significantly improve efficiency, as the Raspberry Pi would be vastly outperformed because, as shown in Fig. II, the Laptop has much more power and

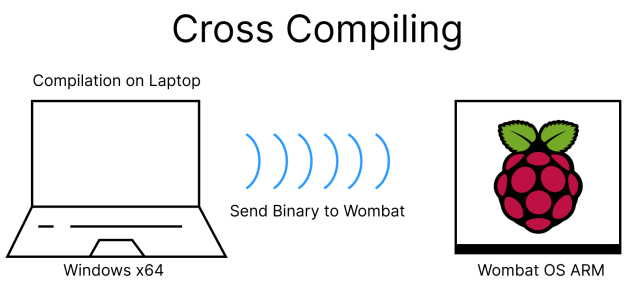

Fig. 1: Cross-compiling

	Raspberry Pi 3B	Intel Core i5 12500H
CPU-Cores	4	12
Clock Speed (Ghz)	1.2	4.5
RAM (GB)	1	16

TABLE II: This table compares the Raspberry Pi 3B with the laptop and its processor [5], which would have been used in cross-compilation

better hardware than the Raspberry Pi 3B. As well as being far more powerful, the binary sent would only need to be executed, which would cut down additional time needed to load or unzip any files.

V. Cross Compilation in Practice

As previously mentioned, the system used for competition is a powerful laptop, which can be seen compared to the Wombat in Fig. II. With Docker, everything needed was transferred using a container. Various programs were then compiled for the appropriate target architecture and lastly, just the binary was sent back as shown in Fig. 2. The authors managed to set up this method successfully, however, various libraries have been compiled into the binary as a ”shared library”. This means that the binary will not run without the correct libraries

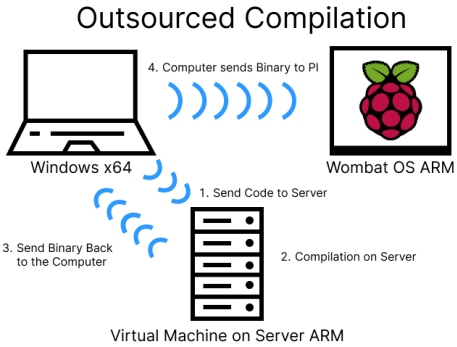

Fig. 2: Distributed Compilation

in the correct directory. You would have to create the same file structure as on the Wombat, which is not possible on Windows. On Linux, it would theoretically be possible to create the file structure with the libraries, but the libraries imported in the project would still not work, so while the binary was compiled successfully, it could not be executed on the Wombat, because shared libraries could not be loaded, and so the authoring effort ended.

VI. Distributed Compilation in Theory

Distributed compilation requires a powerful third-party system, such as a server. This server is expected to receive the programs, compile them and then send them back to the Wombat. To do this, the server requires libraries and programs to compile the code, which the authors solved using images which were run on virtual machines. In this case an image means a disk image of the Wombat. These images act as copies of the data within the Wombat, which would be sent to the server. In this manner, the server can correctly run the program. This should allow for faster compilation time. The server would then compile the code and send the binary to the Wombat, which would only need to be executed on the Raspberry Pi.

VII. Distributed Compilation in Practice

The authors received the server from the robo4you organization, which enabled testing the speed of the server, as well as comparing distributed to native compilation. The server had 16 gigabytes of RAM and 8 CPU cores. To implement the image mentioned above, the instruction manual in the KIPR repository was used [8]. To be able to compile the programs, an image was required that contains all the libraries needed. This image was created locally on a laptop and needed to be sent or copied onto the server. Simply transferring it was not possible, due to there being no transferring protocol such as the secure copy protocol (SCP) [6].

Hosting the image on a web server and having it downloaded using the `wget` [7] command was a possibility, although more problems like the virtual machine crashing by itself after a set amount of time arose.

The Raspberry Pi 3 Model B only contains one gigabyte of RAM, so to make it faster, it was attempted to add more RAM, using partitions on a virtual machine. To be able to change the amount over the maximum of one gigabyte, the QEMU template of the Raspberry Pi would need to be changed. During testing, it turned out that the response and compilation times would be much slower than expected due to the process of sending data back and forth and unfortunately the authors could not measure the pure compile times.

VIII. Experiments

The laptop compared in Fig. II was also used in all the tests. The authors tested the compilation times of multiple, in size different programs, which have the purpose of testing the compilation time on the Raspberry and the computer to see if the results would match the graph in Fig. II and the assumptions about the speed of the Pi. For this purpose, multiple C++ programs with the compiler g++ [11] were executed on both computers. The exact time was then provided by the "time" command [12].*Here is an example of what a command looked like:* `time g++ file.cpp -o test`. The programs were compiled several times on both computers and the average time required was taken, which can be seen in the graph Fig. 4. As can be seen from the graph, the assumption about the speed of the Pi and also the superiority of the computer was confirmed.

Fig. 3: KIPR Controller: Wombat

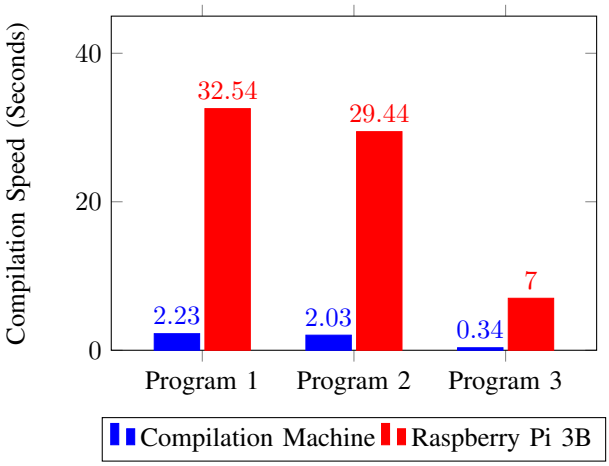

Fig. 4: This diagram compares the Raspberry Pi 3B with the same laptop that is used in Fig. II. Multiple programs were compiled several times on both computers and the average of these results is shown. The first program is that of the second bot and the second that of the mainbot. The third was a small test program.

IX. Results

Cross compilation was successfully set up by the authors, though it had to be scrapped for various reasons. One such reason was the implementation of the libwallaby and third party libraries. These could not be accessed to transfer and use on the authors' laptops, making compiling the code on the computer impossible. Various fixes were tried, however each of them failed, so it was decided to scrap it in exchange for distributed compilation.
The previously mentioned option requires a powerful server, which was provided to the researchers. To fix the previously mentioned problem in cross-compilation, an image was used to copy the data of the Wombat onto the third party computer. While this option managed to work properly for compiling code, it ended up being very likely to crash.

X. Discussion and Conclusion

In conclusion, the implementation of both versions was successful, however, problems which were too time consuming to fix, caused them to be scrapped. Unforeseen circumstances made both options to optimize effectiveness and compilation speeds impossible. For cross-compilation, direct access to the libraries would solve the problem, allowing for faster run-times and compilation. An image of the Wombat would not solve this problem for the same reasons it did not during distributed compilation. So, while cross-compilation did not exactly fail, being unable to use the needed libraries unfortunately forced the authors to completely remove all useability for the botball competition.
In a similar manner, the setting up and implementation of distributed compilation did not fail either. The transfer of files using an image lacked the secure copy protocol or other transfer protocols, which is one of the reasons the response times and compilation times is much slower than native compilation. Even after extensive testing and multiple different approaches, such as static linking, the results ended in failure. Setting up both versions did not cause any troubles, although the results of this paper show that cross-compilation is the superior and better option, due to the lasting problem being a fault on the needed libraries, instead of slower compilation times. The results of said success can be seen in Fig. 4, and the experiment can be replicated by following the instructions in the section "experiments". Should it be able to circumvent the problem of adding the necessary libraries, this choice would be faster. Additionally it would enable usage of various editors and IDEs, such as Virtual Studio Code, while still allowing to build and execute the code on the Wombat. This would not only decrease the time needed to compile programs, but also allow for faster programming thanks to better work environments or using the auto completion function available in various editors. Many of the resources used can be found on the authors' GitHub repository [13] to facilitate future teams' subsequent work.

Acknowledgements

The authors would like to thank Dr. Michael Stifter, Konstantin Lindorfer, and all the other members of robo4you for their invaluable support throughout the creation of this work.

References

[1] V. Mankinen, V. Rahkonen *Cross-Compiling tutorial with Scratchbox* IN: Scratchbox, 2005.
[2] W. Gay *Cross-Compiling* IN: Springer, 2018.
[3] Raspberry Pi 3B *https://www.raspberrypi.com/products/raspberry-pi-3-model-b/* (accessed 2024-02-22)
[4] KIPR Wombat *https://www.kipr.org/kipr/hardware-software* (accessed 2024-02-22)
[5] Intel Core i5 12500H *https://www.intel.de/content/www/de/de/products/sku/96141/intel-core-i512500h-processor-18m-cache-up-to-4-50-ghz/specifications.html* (accessed 2024-02-22)
[6] SCP *https://venafi.com/blog/what-secure-copy-protocol-and-how-use-it/, https://de.wikipedia.org/wiki/Secure_Copy* (accessed 2024-02-23)
[7] wget *https://www.ionos.com/digitalguide/server/configuration/linux-wget-command/* (accessed 2024-02-23)
[8] KIPR GitHub Repository *Creating a Virtual Wombat Image, https://github.com/kipr/qemu-wombat* (accessed 2024-01-30)
[9] Bellard, Fabrice *QEMU, a fast and portable dynamic translator.* IN: USENIX annual technical conference, 2005.
[10] Miell, Ian and Sayers, Aidan *Docker in practice* IN: Simon and Schuster, 2019.
[11] g++ *https://www.geeksforgeeks.org/compiling-with-g-plus-plus/* (accessed 2024-02-23)
[12] time command *https://linuxize.com/post/linux-time-command* (accessed 2024-02-23)
[13] The authors' GitHub repository *https://github.com/WeC0deIT/ Paper_Resources_2024* (created 2024-03-26)