École Polytechnique Fédérale de Lausanne

Migrating Makefiles to the Meson build system

by Mirko Rado

Bachelor Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Expert Reviewer
External Expert

Florian Hofhammer
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 9, 2023

Abstract

Makefiles have long since been the preferred tool for building projects, especially for low level programming languages such as assembly or C. However, their rudimentary syntax and lack of high level functionalities make maintaining large projects a long and complex task, prone to errors. Modern build systems such as Meson help automate some tasks by providing high level functions and by splitting the overall build process in two main steps: the configuration step and the build step. The programmer is only required to define the configuration step, in which the project's dependencies and targets are specified in a simple, easy to understand language. This project aims to migrate the existing Makefiles of EmbedFuzz to Meson, providing a flexible, maintainable and simple build system. The result is a concise and modular build configuration outperforming GNU Make by a second and a half to two seconds on average build wall time, or a 25 to 35% improvement, and proving to be more reliable in detecting target dependencies when re-building after some file modifications.

Résumé

Les Makefiles sont depuis longtemps l'outil de prédilection pour la construction de projets, en particulier pour les projets à langages de programmation de bas niveau tels que l'assembleur ou le C. Cependant, leur syntaxe rudimentaire et leur manque de fonctionnalités de haut niveau font de la maintenance de grands projets une tâche longue et complexe, sujette aux erreurs. Les systèmes de construction modernes tels que Meson aident à automatiser certaines tâches en fournissant des fonctions de haut niveau et en divisant le processus de construction global en deux étapes principales : l'étape de configuration et l'étape de construction. Le programmeur n'a qu'à définir l'étape de configuration, dans laquelle les dépendances et les cibles du projet sont spécifiées dans un langage simple et facile à comprendre. Ce projet vise à migrer les Makefiles existants d'EmbedFuzz vers Meson, offrant une solution flexible, maintenable et un système de construction simple. Le résultat est une configuration de construction concise et modulaire surpassant GNU Make d'une seconde et demie à deux secondes en moyenne sur le temps de construction, ce qui représente une amélioration de 25 à 35%, et Meson s'est révélé plus fiable dans la détection des dépendances cibles lors de la reconstruction après quelques modifications de fichiers.

Contents

Chapter 1

Introduction

EmbedFuzz is a multi-language project developed by the HexHive [5] lab at EPFL aiming to enable high performance fuzz testing for embedded systems' firmware. EmbedFuzz uses a custom built loader, specific to each firmware, to load the firmware's corresponding binary to the virtual environment and run the fuzzing campaign. Our work consists in using Meson to build this firmware-specific loader and link it with the necessary files for rewriting the firmware binaries.

Since GNU Makefiles specify the build targets by hand using a rudimentary syntax, the first of many challenges to migrating EmbedFuzz's existing Makefiles resides in explicating the order of the build targets and their multiple dependencies. The second consists in abstracting the build into a configuration step: refraining from explicitly writing the targets but specifying what Meson needs to be able automatically generate them. The third and last challenge is the establishment of a modular design, evicting any hard-coded paths and facilitating the potential inclusion of additional firmware.

The existing build system for EmbedFuzz is difficult to understand for anyone not familiar with the project. The Makefiles, by inherent design, contain a mix of target definitions, environment variables, help messages, explicit dependency handling and overwriting of default values. This approach is hard to maintain. Some details are easy to miss and errors are consequently inserted in the code resulting in long hours of debugging as build systems do not provide many tools for this purpose. Since building the project has little to do with the project itself and has no concrete interest, we would like to spend as little time as possible setting up the build. Therefore, Meson's high level configuration approach is preferable.

To approach the problem it is best to first specify the requirements: we want a build system that is both simple and flexible. There is no simplicity without high level abstraction and there is no flexibility without modularity. Conveniently, we can combine both using Meson. Indeed, Meson abstracts away the target specification details and provides the user with a python-like custom language for specifying the configuration files [7]. Python emphasizes code readability and simplicity, making it an excellent choice for beginners [8]. Its clean and intuitive syntax allows developers to express concepts in fewer lines of code compared to other languages, promoting productivity and reducing development time. Thus Meson configuration files are quick and simple to write.

Simplicity, flexibility and performance can be combined when building large projects.

One might believe there must be trade-off with performance given the aforementioned constraints. However, our benchmark results point to the opposite. We compared the performance of the handwritten Makefiles and the auto-generated Ninja back-end build files (the configuration step is not used in our performance tests as Make does not provide a configuration step [6]) and found, given a maximum number of available CPU cores, that Ninja outperformed Make by over a second and a half to two seconds on average wall time. In addition, Meson proved to be more reliable than Make as it was more sensitive to file modification than Make. Indeed, Meson handles dependencies automatically while Makefile dependencies rely on the programmer and are therefore prone to human errors.

Chapter 2

Background

There are two discrete main steps in the overall Meson build process [7]: the configuration step and the build step. The configuration step is written by the programmer to specify project dependencies, the hardware it will build on and the different targets to build. Meson then generates the build files and executes them during the build step. This two step approach is used in other popular build systems too such as CMake [1] and GNU Autotools [3]. This design similarity provides a certain degree of compatibility between these build systems and Meson even provides a functionality to automate the migration from CMake or GNU Autotools.

Meson is designed to be user-friendly without sacrificing performance. Hence, the configuration step is explicitly defined in meson.build files using a custom language similar to that of the Python programming language, which is considered easy to understand even by people who have never programmed in Python [8]. The meson.build files specify the project name, language and compiler inherent to the project, but also the dependencies (such as libraries, files, executable programs), the targets and build options (compiler and linker flags for instance).

Meson organizes projects into a tree-like structure with the root directory of the project containing the top level meson.build file, which itself can invoke Meson sub-projects or other meson.build configuration files in sub-directories. All variables defined in top-level meson.build files are accessible from the sub-directory. This is ideal for modularity as it enables the top-level configuration file to be generic when building multiple targets by overwriting and/or modifying variables.

Furthermore, it is also possible to define multiple architectures to build on. Meson allows its users to specify one native architecture and one cross build architecture per sub-project. Both architectures are specified in separate text files containing their respective system specifications along with the compiler and linker to use for the build. Other important properties such as additional compiler or linker flags can be added as well and accessed easily within Meson's configuration files. However, Meson only allows for one host and one target architecture per project or sub-project.

Lastly, global target options can be specified in a text file. Each option is assigned a name, a type and a default value. Options can be accessed similar to architecture properties in Meson's configuration files but their values must be set from the command line before the generation of the build files. Build options are set at configuration time not during build time and are used to determine how to build the project: enabling/disabling features such as coverage or testing and setting the output directory.

The programmer then calls `meson setup` from the command line, providing the above cross and native files as arguments to initialize the Meson build directory. Entering this newly initialized directory, the programmer can configure it further notably by specifying options from the options file before building the project.

Chapter 3

Design

We already stated that the goal of the project was to make a modular build system. To this effect, we designed the top-level configuration file, located at the root of the project, to be as generic as possible. The natural design therefore consists in regrouping similar properties, targets and attributes in the top-level configuration file and any firmware-specific details in the corresponding firmware sub-directory configuration file. Paths are not hard-coded but relative to the source and build roots of the project determined by using Meson's built-in functionalities.

The loader is defined in the source directory of the project and must be built on a native ARM machine with a particular GNU compiler. Hence, the specifications of the native architecture are outlined in a native file and used when defining the target for the loader. If the firmware binary does not yet exist, it needs to be cross compiled since we use a different compiler and linker and different flags. These system specifications are defined in the corresponding cross file.

Each firmware can be rewritten with a coverage report or without. This is an option the user can set from the command line. Hence coverage options and the different firmware to be built are enumerated in an options text file and specified with the command line before running the compilation and the auto-generating the build files.

Chapter 4

Implementation

It is important to understand Meson creates a distinction between the source directory and the build directory [7]. The source directory (along with the root directory) is used to store the code of the project and contains the configuration files for Meson to work with. The build directory is used for all files auto-generated by Meson and used in the build process. This distinction is a design choice by Meson's developers and must be respected. This design choice notably makes it possible to setup multiple build directories under various configurations or options while in-source build systems require a reconfiguration at every build. Build related and auto-generated files do not pollute the source directory either and cleaning the build is made easy by simply removing the build directory [9].

EmbedFuzz is a multi-directory project, containing a `firmware` directory with all the firmware to be tested using the fuzzing technique. Most of the firmware already contain their corresponding binary executable and the executable is copied to the build directory for Meson to overwrite. However, some firmware binaries need to be built from C source code. We define them as `executable` targets and specify their dependencies (this is done in the respective sub-directory configuration files, see example listing 4.1). Meson will by default use cross compilers from the cross file when they are available. All firmware need to be rewritten using a custom loader defined in the `src` directory. This loader, written in C, must be compiled on native architecture and is firmware specific. The configuration file of the loader is located in the `src` directory and invoked from the root-level configuration file.

Listing 4.1: Defining the binary executable for the cov-test firmware

```
exe = executable('cov-test', c_files, c_args : c_flags, link_args : c_flags)
```

Modularity in the build configuration of the project is closely related to the project's tree-like structure. Meson provides a `subdir` function invoking Meson configuration files from sub-directories as shown in listing 4.2. This enables modularity by specifying firmware-specific configuration details in their corresponding directory or defining intermediate targets in lower Meson configuration files. As a result, the top level configuration file may be as generic as possible and hence will likely not require further modification. All navigation through the project's tree-like structure is relative to the root build and source directories, which conveniently means the user is only required to maintain the project's inner structure and does not have to worry about external directories or program files. Meson's built-in functionalities will automatically search for necessary programs given the user provides the program name. This makes our build system not only robust to a change of location of the project in the file system, but the configuration of the build is also indifferent to external modifications of the file system and platform-independent (Meson detects the underlying operating system).

Listing 4.2: Invoking the loader's configuration file

```
    # Build the loader for the firmware
    subdir('src/loader')
```

Meson is a tool that supports multiple languages, including C. However, it cannot account for all languages or commands. For such targets we use Meson `custom_target` and specify the command to run along with any input dependencies similar to executable targets in C. The `custom_target` command is used in EmbedFuzz to define the targets invoking Python scripts for auto-generating yaml files, C code or rewriting the firmware binaries as shown in the listing 4.3 below. Here we set the `build_by_default` variable to true as this is the main target of the project and should always be re-built when any source file changes. The `@INPUT@` symbols refer to elements from the input `deps` array. The name of the output file is the same as the target name with `-rewritten` appended to it.

Listing 4.3: Rewriting the firmware target

```
custom_target(target,
    input : deps,
    output : target + '-rewritten',
    command : [prog_python, '-m', 'rewriter',
                            '-t', '@INPUT0@',
                            '-i', '@INPUT1@', coverage, '-e',
                            '-s', '@INPUT2@',
                            '-b', '@INPUT3@',
                            '@INPUT4@', '@OUTPUT@'],
    build_by_default : true
)
```

Chapter 5

Evaluation

Firstly, it must be noted that Meson takes care of the configuration step only. That is to say, the programmer defines the configuration of the build in `meson.build` files which Meson will then translate to build.ninja files containing the build targets to be executed by the Ninja back-end. Therefore, as a first evaluation metric between Meson and Make we compare the number of files and the number of lines of code written for each, as well as the number of changes required when adding a new firmware to be built.

In this regard, our Meson configuration has a total of 18 files (including the cross, native and options files) for 162 lines of code, most of which are contained in the general firmware configuration file and the one for the loader. Adding a firmware with an existing binary would only require adding a `meson.build` file with a single line of code copying the binary to the build directory for Meson to work with and adding the firmware's name to the list of existing firmware in the options file. Without an existing binary, four lines of code would be needed instead of one. The previous Make build system was constituted of 17 files for a total of 268 lines of code. Adding a firmware incurs an additional 11 lines of code should the firmware binary already exist, 16 on the other hand if there is a need to build it. While this is a significant difference, the additional lines of code for Make are similar to other firmware and can be rapidly added. However, this similarity stems from code duplication which in general is bad practice and should be avoided for good maintainability and to limit security vulnerabilities [2].

As a second evaluation metric, we compare the run-time of building the loader of each firmware using the Meson front-end `meson compile` command, the `ninja` back-end command, the default single-threaded `make` command and the multi-threaded `make -j $(nproc)` command. We only compare the run-time of building the loader and not rewriting the entire firmware as the loader is the only target entirely written in C code which does not invoke Python scripts. Thus measuring the build time of the loader corresponds to the genuine performance of our build systems and eliminates the overhead of running Python scripts which might falsify our results. The run-times of the `setup` and `configuration` phases to appropriately setup the build directory and configure the firmware (we always run with coverage enabled) are not taken into account within the back-end tests as these are part of the configuration step and not the build step.

There are three different kinds of measurements: building the loader from scratch, re-building the loader when the binary already exists and re-building the loader after modifying a file. This last measurement also accounts for the reliability of the system to establish the correct dependencies between the targets and ensure the system always re-builds to the latest version of the code.

All benchmarks are run on an Asus Rog Zephyrus laptop with Ryzen 7 (6000 series) multi-processors, 16GB of RAM, running within a Docker container based on Ubuntu 22 leveraging QEMU user to emulate the ARM architecture. Each firmware is run under the same conditions for 10 iterations for statistical significance. The same benchmarks are run on the native ARM machines, exhibiting similar results only without the overhead of emulation, the details of which can be found in Appendix B.

Figure 5.1 shows a wall time comparison in seconds for each firmware with respect to, from left to right, Meson using the front-end `meson compile` command completely deleting the build directory between measurements, Ninja using the `ninja` command also removing the build directory, Make using `make -j $(nproc)` and finally the `ninja` command using `meson -wipe` to wipe the build directory between measurements. Figure 5.2 shows the same statistics for CPU time.

Our results show using Meson's built-in functionality for wiping the build directory is considerably faster than removing it completely. Indeed, combining the direct use of Ninja's back-end command with Meson's wipe functionality outperforms Make in both wall time and CPU time on every firmware, despite Ninja having some outliers. Curiously enough, the Meson's front-end command, while being functionally equivalent to the Ninja back-end command, incurs a one second performance overhead.

This overhead is consistent over all benchmarks we made and is more notable when re-building with no work to be done. Meson's front end command takes around a second to realise there is nothing to be done while both Ninja and Make take around 5 hundredths of a second, with Ninja being on average 3 milliseconds faster. After modifying one of the loader's source files however, Make assures us there is no work to be done while Meson and Ninja re-build the corresponding target (taking less time than re-building from scratch). This is likely an error on behalf of the programmer who missed a dependency check when designing the Makefiles. For full details, see Appendix A.

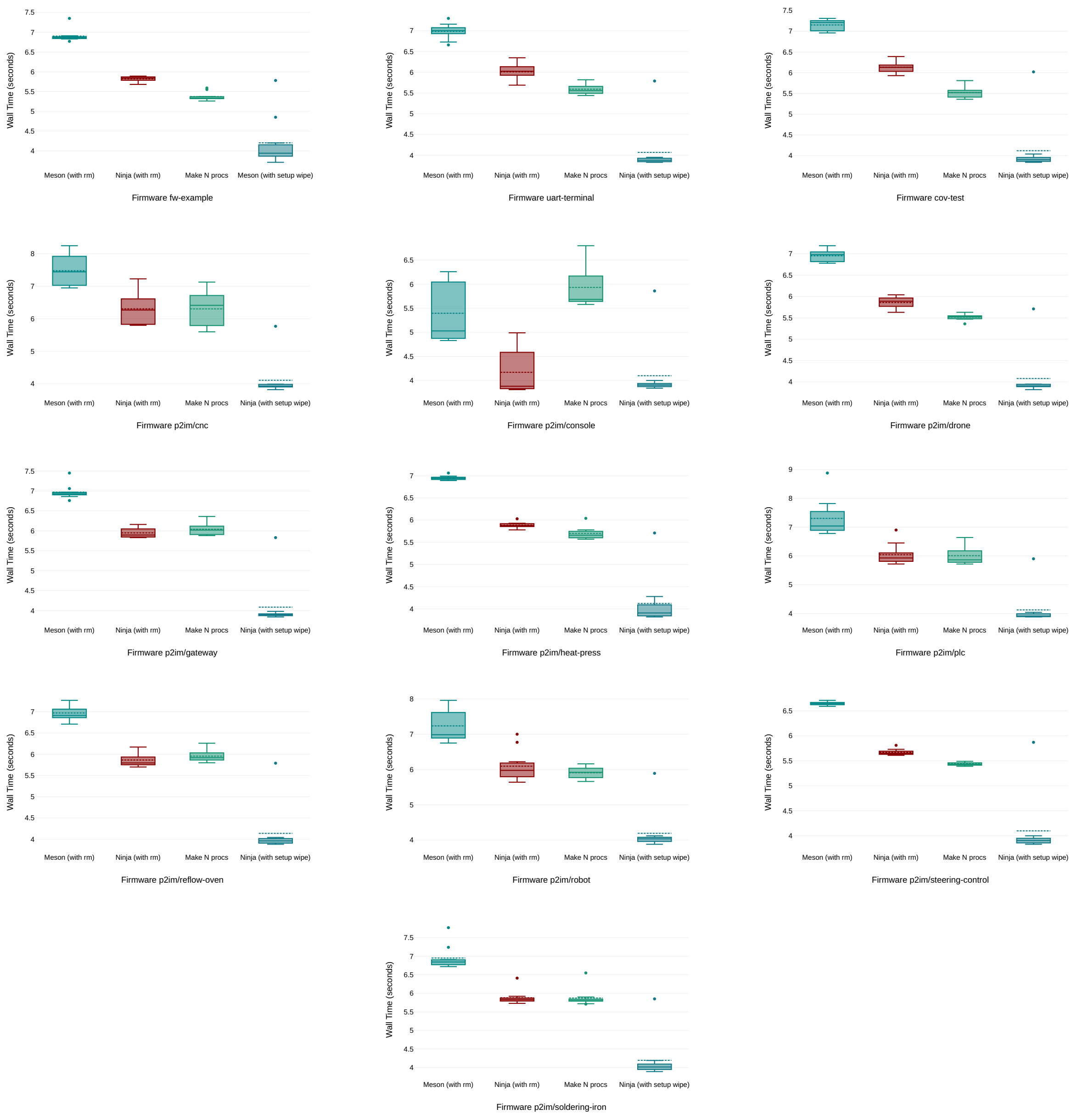

Figure 5.1: Wall time comparison when rebuilding from scratch

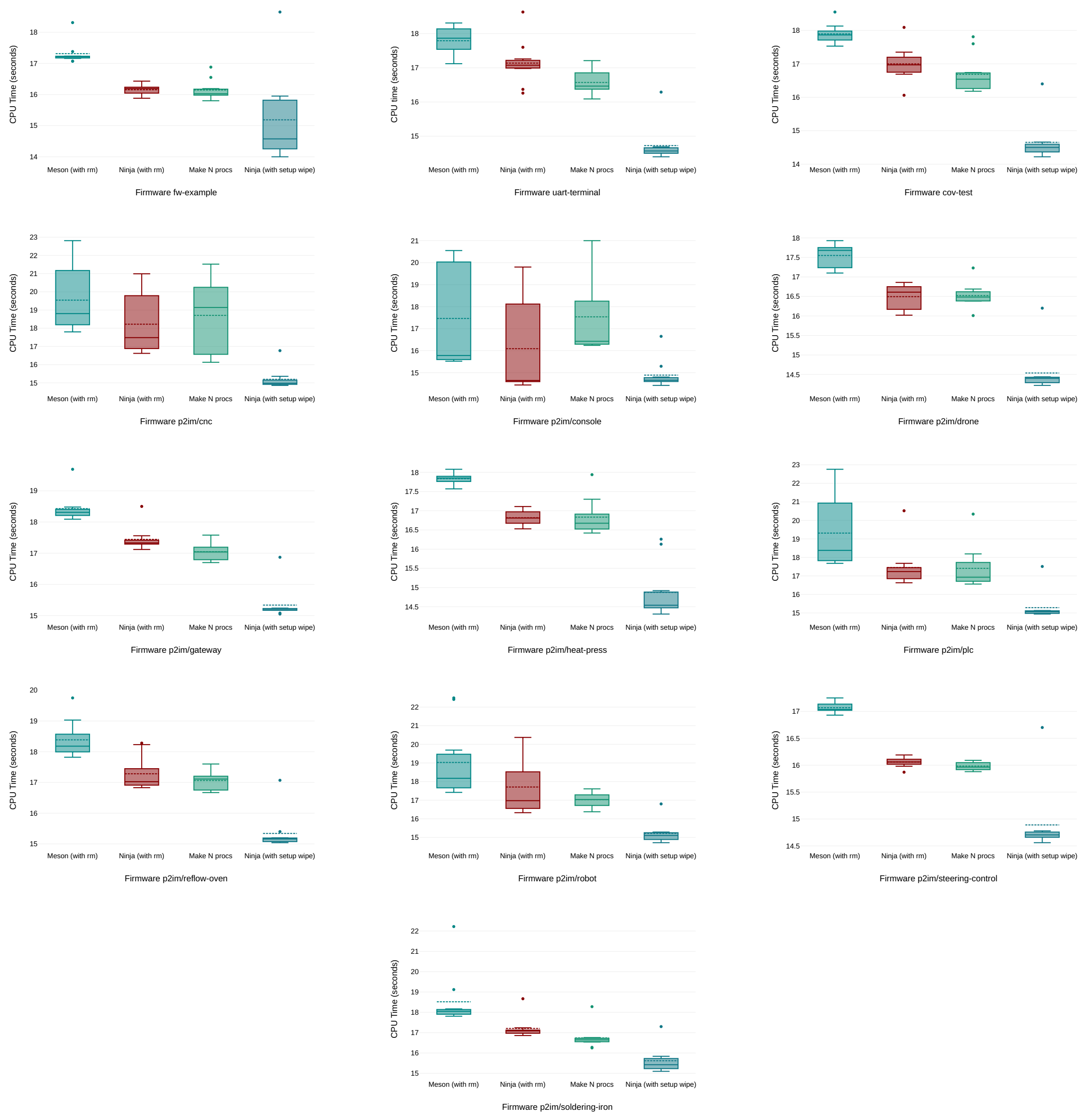

Figure 5.2: CPU time comparison when rebuilding from scratch

Chapter 6

Related Work

Apart from Meson and Make, there are several other build systems available that are commonly used in software development. Some are specific to certain programming languages like Gradle for Java projects [4] and others can support multiple languages. For instance, CMake is a popular cross-platform build system that generates platform-specific build files (e.g., Makefiles, Visual Studio projects) based on a high-level CMakeLists.txt configuration [1]. It supports various programming languages and can be used to build projects on different platforms. Another is SCons, a software construction tool written in Python. It uses Python scripts as build configuration files, which allows developers to have full programming capabilities while defining the build process. SCons supports multiple programming languages and platforms [10].

Other build systems in the Linux and Unix ecosystems include GNU Autotools, particularly useful for projects that need to be built on multiple platforms and require extensive configuration and dependency management. It provides a standardized way to handle system differences and create portable build systems. However, it's worth noting that Autotools can have a steeper learning curve compared to some other build systems, as it involves writing complex configuration files and using custom macros [3]. Additionally, the Autotools ecosystem has seen some criticism for its complexity and the learning overhead involved. As a result, alternative build systems like CMake and Meson have gained popularity in recent years for their simpler syntax and easier learning curve. Nevertheless, GNU Autotools remains a powerful and widely used build system, especially in the Unix and Linux communities.

Meson provides a benchmark comparison against GNU Autotools, CMake and SCons. The results show Meson outperforming its rivals for cumulative configuration, build and empty build times[7].

Chapter 7

Conclusion

As we have seen, building large scale projects significantly increases the complexity and number of the build targets involved, thus making writing build files by hand a difficult task. High level build systems such as Meson offer the programmer high level functionalities and abstractions, yielding readable modular build configurations providing both flexibility and performance. A lower number of code lines, increased simplicity and maintainability and a performance improvement of over a second make Meson a preferable tool to Make for such large projects.

Bibliography

[1] "CMake documentation". In: URL: `https://cmake.org/documentation/`.

[2] "Duplicate Code: Everything you need to know". In: URL: `https://www.codegrip.tech/productivity/what-is-duplicate-code/`.

[3] "GNU Autotools". In: URL: `https://www.gnu.org/software/automake/manual/html_node/index.html`.

[4] "Gradle documentation". In: URL: `https://gradle.org/`.

[5] "HexHive web page". In: URL: `https://hexhive.epfl.ch/`.

[6] "Make documentation". In: URL: `https://www.gnu.org/software/make/manual/make.html`.

[7] "Meson documentation". In: URL: `https://mesonbuild.com/`.

[8] Hillary Nyakundi. "Why Python is Good for Beginners – and How to Start Learning It". In: 2023. URL: `https://www.freecodecamp.org/news/why-learn-python-and-how-to-get-started/`.

[9] Jussi Pakkanen. "Why you should consider using separate build directories". In: URL: `http://web.archive.org/web/20190715081007/http://voices.canonical.com/jussi.pakkanen/2013/04/16/why-you-should-consider-using-separate-build-directories/`.

[10] "SCons: A software construction tool". In: URL: `https://scons.org/`.

Appendix A

Additional measurements on ASUS laptop

The maximum, minimum and average were computed over 10 measurements for statistical significance.

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	1.189	1.022	1.066	0.057	0.053	0.055	0.061	0.058	0.060
uart-terminal	1.081	1.014	1.049	0.056	0.053	0.055	0.058	0.055	0.057
cov-test	1.063	1.016	1.031	0.056	0.053	0.054	0.061	0.058	0.059
p2im/cnc	1.049	1.011	1.029	0.054	0.051	0.052	0.058	0.056	0.056
p2im/console	1.049	1.011	1.036	0.054	0.052	0.053	0.057	0.055	0.056
p2im/drone	1.086	1.038	1.057	0.056	0.053	0.055	0.060	0.055	0.058
p2im/gateway	1.168	1.027	1.078	0.060	0.052	0.055	0.066	0.056	0.059
p2im/heat-press	1.133	1.077	1.103	0.057	0.053	0.055	0.060	0.056	0.058
p2im/plc	1.113	1.054	1.081	0.055	0.052	0.054	0.060	0.056	0.058
p2im/reflow-oven	1.104	1.053	1.072	0.055	0.051	0.053	0.059	0.056	0.058
p2im/robot	1.106	1.053	1.081	0.055	0.053	0.054	0.059	0.056	0.057
p2im/soldering-iron	1.285	1.036	1.126	0.057	0.053	0.054	0.058	0.056	0.057
p2im/steering-control	1.126	1.060	1.087	0.056	0.052	0.054	0.059	0.056	0.058

Table A.1: Wall time (in seconds) when no work to be done

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	1.194	1.027	1.071	0.056	0.053	0.054	0.061	0.057	0.059
uart-terminal	1.086	1.019	1.054	0.056	0.052	0.054	0.058	0.055	0.056
cov-test	1.068	1.020	1.036	0.055	0.052	0.053	0.060	0.057	0.058
p2im/cnc	1.054	1.015	1.034	0.053	0.050	0.052	0.057	0.055	0.056
p2im/console	1.054	1.015	1.040	0.054	0.051	0.053	0.057	0.055	0.056
p2im/drone	1.090	1.042	1.061	0.055	0.053	0.054	0.059	0.055	0.057
p2im/gateway	1.168	1.030	1.081	0.060	0.051	0.054	0.065	0.056	0.058
p2im/heat-press	1.138	1.083	1.107	0.056	0.053	0.054	0.059	0.055	0.057
p2im/plc	1.117	1.059	1.085	0.054	0.051	0.053	0.059	0.056	0.058
p2im/reflow-oven	1.108	1.058	1.076	0.054	0.051	0.053	0.058	0.056	0.057
p2im/robot	1.111	1.057	1.085	0.055	0.052	0.053	0.058	0.055	0.057
p2im/soldering-iron	1.291	1.040	1.131	0.055	0.052	0.053	0.058	0.055	0.057
p2im/steering-control	1.130	1.064	1.091	0.055	0.052	0.053	0.059	0.056	0.058

Table A.2: CPU time (in seconds) when no work to be done

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	3.028	2.905	2.944	2.005	1.870	1.929	0.062	0.059	0.060
uart-terminal	3.045	2.938	3	2.039	1.901	1.964	0.059	0.056	0.057
cov-test	3.097	2.866	2.959	2.105	1.861	1.929	0.063	0.059	0.060
p2im/cnc	3.117	2.917	2.964	1.999	1.897	1.947	0.062	0.056	0.058
p2im/console	3.087	2.927	2.984	1.959	1.900	1.922	0.059	0.056	0.057
p2im/drone	3.098	2.912	3.009	1.979	1.902	1.930	0.059	0.056	0.057
p2im/gateway	3.018	2.906	2.961	1.956	1.893	1.920	0.059	0.056	0.058
p2im/heat-press	3.102	2.905	3.007	1.974	1.902	1.944	0.062	0.057	0.059
p2im/plc	3.249	2.937	3.057	2.127	1.904	2.000	0.063	0.056	0.058
p2im/reflow-oven	3.253	2.930	3.035	2.030	1.873	1.951	0.060	0.056	0.058
p2im/robot	3.148	2.850	2.978	2.107	1.863	1.947	0.062	0.056	0.058
p2im/soldering-iron	3.383	2.862	3.009	2.161	1.867	1.944	0.062	0.056	0.058
p2im/steering-control	3.541	2.936	3.334	2.269	1.876	2.115	0.066	0.058	0.063

Table A.3: Wall time (in seconds) when re-building after modifying a source file

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	3.038	2.915	2.954	2.009	1.874	1.933	0.061	0.058	0.060
uart-terminal	3.053	2.948	3.009	2.041	1.905	1.968	0.058	0.056	0.057
cov-test	3.105	2.875	2.968	2.109	1.865	1.933	0.063	0.058	0.060
p2im/cnc	3.126	2.927	2.973	2.004	1.901	1.952	0.061	0.055	0.057
p2im/console	3.096	2.936	2.993	1.964	1.904	1.926	0.057	0.056	0.057
p2im/drone	3.107	2.921	3.018	1.983	1.906	1.934	0.058	0.055	0.057
p2im/gateway	3.028	2.915	2.971	1.960	1.895	1.924	0.058	0.055	0.057
p2im/heat-press	3.111	2.913	3.016	1.978	1.908	1.948	0.061	0.056	0.058
p2im/plc	3.259	2.946	3.065	2.124	1.908	2.004	0.062	0.055	0.057
p2im/reflow-oven	3.257	2.939	3.043	2.034	1.877	1.954	0.059	0.055	0.057
p2im/robot	3.159	2.860	2.988	2.111	1.867	1.951	0.062	0.055	0.057
p2im/soldering-iron	3.389	2.871	3.017	2.164	1.872	1.948	0.061	0.055	0.057
p2im/steering-control	3.541	2.945	3.339	2.271	1.879	2.117	0.066	0.057	0.062

Table A.4: CPU time (in seconds) when re-building after modifying a source file

Appendix B

Additional measurements on native ARM machines

The maximum, minimum and average were computed over 10 measurements for statistical significance.

The native ARM machines are NXP Layerscape LX2160A 16-core Arm Cortex A72, with 32GB RAM running Ubuntu 22.04.

	Meson with rm			Ninja with rm			Make single proc			Make N procs			Ninja with wipe		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	4.27	3.65	3.90	3.13	2.84	3.00	5.50	5.04	5.24	2.45	2.21	2.33	3.11	1.82	2.05
uart-terminal	4.01	3.60	3.79	3.11	2.86	2.95	5.50	5.02	5.24	2.65	2.23	2.38	3.17	1.88	2.06
cov-test	3.92	3.72	3.81	3.26	2.91	3.04	5.51	5.02	5.20	2.56	2.16	2.32	2.94	1.84	2.03
p2im/cnc	4.13	3.55	3.70	3.08	2.79	2.94	5.67	5.13	5.34	2.76	2.38	2.53	2.96	1.87	2.06
p2im/console	2.95	2.58	2.71	1.94	1.78	1.88	5.48	5.19	5.28	2.63	2.34	2.44	3.02	1.82	2.02
p2im/drone	3.93	3.59	3.71	3.27	2.84	3.03	5.19	4.97	5.11	2.51	2.24	2.35	3.07	1.86	2.06
p2im/gateway	4.00	3.60	3.77	3.19	2.86	3.02	5.64	5.31	5.44	2.65	2.48	2.55	3.09	1.86	2.06
p2im/heat-press	3.96	3.56	3.75	3.32	2.88	3.01	5.35	5.01	5.24	2.50	2.26	2.39	2.78	1.85	2.00
p2im/plc	3.99	3.61	3.77	3.14	2.87	3.00	5.55	5.25	5.38	2.61	2.35	2.49	2.93	1.85	2.04
p2im/reflow-oven	3.78	3.56	3.68	3.15	2.88	2.95	5.52	5.21	5.34	2.70	2.37	2.51	3.00	1.85	2.07
p2im/robot	3.93	3.63	3.78	3.28	2.83	3.04	5.56	5.16	5.30	2.68	2.38	2.49	3.17	1.85	2.04
p2im/soldering-iron	4.07	3.51	3.83	3.32	2.84	3.05	5.56	5.29	5.42	2.75	2.51	2.61	2.99	1.86	2.08
p2im/steering-control	4.02	3.50	3.76	3.12	2.85	2.94	5.52	5.05	5.23	2.40	2.29	2.35	2.88	1.84	2.03

Table B.1: Wall time (in seconds) when building from scratch

	Meson with rm			Ninja with rm			Make single proc			Make N procs			Ninja with wipe		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	7.17	6.42	6.76	6.18	5.53	5.85	5.49	5.03	5.23	5.75	5.30	5.59	6.06	4.59	4.93
uart-terminal	6.93	6.45	6.67	6.06	5.67	5.85	5.50	5.01	5.23	5.99	5.32	5.60	5.94	4.79	4.98
cov-test	6.91	6.55	6.68	6.02	5.75	5.88	5.50	5.02	5.20	5.73	5.26	5.52	5.80	4.60	4.84
p2im/cnc	7.32	6.50	6.78	6.19	5.72	6.02	5.66	5.12	5.32	6.06	5.39	5.68	5.98	5.00	5.19
p2im/console	5.98	5.55	5.74	5.06	4.68	4.88	5.47	5.18	5.27	5.89	5.41	5.63	6.06	4.78	4.99
p2im/drone	6.77	6.30	6.56	6.23	5.63	5.89	5.18	4.96	5.11	5.71	5.39	5.55	6.02	4.69	4.96
p2im/gateway	7.35	6.65	6.97	6.40	5.99	6.22	5.63	5.30	5.43	6.02	5.50	5.72	6.56	4.92	5.35
p2im/heat-press	7.00	6.29	6.67	6.42	5.74	5.98	5.34	5.00	5.22	5.88	5.35	5.62	5.66	4.72	4.90
p2im/plc	7.22	6.55	6.89	6.16	5.87	6.05	5.54	5.24	5.37	5.89	5.41	5.65	6.05	4.88	5.15
p2im/reflow-oven	7.03	6.58	6.79	6.26	5.89	6.09	5.51	5.19	5.32	6.03	5.45	5.70	6.17	4.97	5.26
p2im/robot	7.02	6.56	6.81	6.52	5.64	6.12	5.55	5.15	5.29	5.90	5.50	5.73	6.31	4.79	5.05
p2im/soldering-iron	7.33	6.53	7.00	6.47	5.87	6.17	5.55	5.28	5.41	6.04	5.70	5.86	6.10	4.90	5.25
p2im/steering-control	7.06	6.36	6.70	6.24	5.65	5.85	5.51	5.04	5.22	5.72	5.42	5.52	5.79	4.68	4.96

Table B.2: CPU time (in seconds) when building from scratch

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	0.891	0.724	0.767	0.013	0.011	0.012	0.008	0.007	0.008
uart-terminal	0.776	0.689	0.736	0.014	0.011	0.012	0.008	0.007	0.007
cov-test	0.894	0.730	0.766	0.013	0.011	0.012	0.008	0.007	0.007
p2im/cnc	0.890	0.691	0.751	0.015	0.011	0.012	0.010	0.007	0.008
p2im/console	0.872	0.704	0.761	0.013	0.011	0.012	0.008	0.007	0.007
p2im/drone	0.926	0.718	0.776	0.013	0.011	0.012	0.009	0.007	0.007
p2im/gateway	0.856	0.675	0.760	0.013	0.011	0.012	0.008	0.007	0.007
p2im/heat-press	0.768	0.695	0.737	0.014	0.011	0.012	0.008	0.007	0.007
p2im/plc	0.845	0.702	0.765	0.013	0.011	0.012	0.008	0.007	0.008
p2im/reflow-oven	0.772	0.704	0.747	0.012	0.011	0.012	0.008	0.007	0.007
p2im/robot	0.826	0.720	0.758	0.013	0.011	0.012	0.008	0.006	0.007
p2im/soldering-iron	0.845	0.694	0.742	0.013	0.011	0.012	0.008	0.006	0.007
p2im/steering-control	0.919	0.690	0.771	0.014	0.011	0.012	0.008	0.007	0.007

Table B.3: Wall time (in seconds) when no work to be done

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	0.891	0.724	0.767	0.013	0.011	0.012	0.007	0.006	0.007
uart-terminal	0.777	0.689	0.736	0.013	0.011	0.011	0.007	0.006	0.007
cov-test	0.894	0.730	0.766	0.013	0.011	0.012	0.008	0.006	0.007
p2im/cnc	0.890	0.691	0.752	0.014	0.010	0.012	0.008	0.006	0.007
p2im/console	0.872	0.704	0.761	0.013	0.011	0.012	0.007	0.006	0.007
p2im/drone	0.926	0.719	0.776	0.013	0.011	0.011	0.009	0.006	0.007
p2im/gateway	0.856	0.675	0.760	0.012	0.010	0.011	0.007	0.006	0.007
p2im/heat-press	0.768	0.695	0.737	0.013	0.010	0.012	0.008	0.006	0.007
p2im/plc	0.845	0.702	0.765	0.013	0.011	0.012	0.007	0.006	0.007
p2im/reflow-oven	0.772	0.705	0.747	0.012	0.011	0.011	0.007	0.006	0.007
p2im/robot	0.826	0.720	0.758	0.012	0.010	0.011	0.007	0.006	0.007
p2im/soldering-iron	0.844	0.693	0.742	0.012	0.011	0.011	0.007	0.006	0.007
p2im/steering-control	0.919	0.690	0.771	0.013	0.010	0.012	0.007	0.006	0.007

Table B.4: CPU time (in seconds) when no work to be done

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	2.248	1.976	2.049	1.563	1.217	1.299	0.008	0.007	0.008
uart-terminal	2.119	1.889	1.991	1.462	1.154	1.288	0.008	0.006	0.007
cov-test	2.275	1.993	2.074	1.543	1.221	1.315	0.008	0.007	0.007
p2im/cnc	2.218	1.928	2.049	1.270	1.221	1.249	0.007	0.007	0.007
p2im/console	2.341	1.895	2.041	1.299	1.192	1.252	0.008	0.007	0.007
p2im/drone	2.136	1.972	2.034	1.473	1.236	1.312	0.008	0.007	0.007
p2im/gateway	2.234	1.892	2.006	1.483	1.223	1.322	0.008	0.007	0.007
p2im/heat-press	2.304	2.000	2.098	1.529	1.258	1.339	0.008	0.007	0.008
p2im/plc	2.185	1.872	2.030	1.443	1.202	1.279	0.008	0.006	0.007
p2im/reflow-oven	2.167	1.991	2.072	1.452	1.231	1.305	0.009	0.007	0.008
p2im/robot	2.218	1.991	2.054	1.348	1.222	1.292	0.008	0.006	0.007
p2im/soldering-iron	2.230	1.912	2.014	1.296	1.190	1.252	0.008	0.006	0.007
p2im/steering-control	2.308	1.964	2.122	1.505	1.236	1.340	0.009	0.007	0.008

Table B.5: Wall time (in seconds) when re-building after modifying a source file

	Meson			Ninja			Make		
Firmware	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**	**Max**	**Min**	**Average**
fw-example	2.248	1.977	2.049	1.561	1.216	1.299	0.008	0.006	0.007
uart-terminal	2.120	1.889	1.992	1.462	1.154	1.277	0.008	0.006	0.007
cov-test	2.275	1.994	2.073	1.542	1.220	1.315	0.007	0.006	0.007
p2im/cnc	2.217	1.929	2.050	1.269	1.222	1.249	0.007	0.006	0.007
p2im/console	2.187	1.896	2.025	1.298	1.192	1.252	0.008	0.006	0.007
p2im/drone	2.136	1.973	2.034	1.473	1.236	1.298	0.007	0.006	0.007
p2im/gateway	2.234	1.892	2.006	1.483	1.223	1.322	0.007	0.006	0.007
p2im/heat-press	2.300	2.000	2.098	1.528	1.258	1.339	0.008	0.006	0.007
p2im/plc	2.185	1.872	2.030	1.442	1.201	1.279	0.007	0.006	0.007
p2im/reflow-oven	2.167	1.992	2.072	1.452	1.230	1.305	0.008	0.006	0.007
p2im/robot	2.218	1.991	2.054	1.348	1.222	1.292	0.007	0.006	0.007
p2im/soldering-iron	2.231	1.912	2.014	1.296	1.191	1.252	0.007	0.006	0.006
p2im/steering-control	2.308	1.963	2.121	1.504	1.235	1.339	0.008	0.006	0.007

Table B.6: CPU time (in seconds) when re-building after modifying a source file