Efficient Task Scheduling in Build Systems

Arav Khanna[1]

Supervisors: Soham Chakraborty[1]**, Dennis Sprokholt**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2024

Name of the student: Arav Khanna
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Burcu Ozkan

An electronic version of this paper can be found at repository.tudelft.nl.

Abstract

Build systems are essential tools for compiling codebases of any complexity. In order to maximize performance, they use parallelism to complete multiple build steps simultaneously. In this thesis, we examine the effectiveness with which common build systems distribute work across available CPUs. We design an automatic process for fetching, compiling, and inspecting the build steps of common C/C++ codebases, and use it to benchmark various build systems. Based on empirical performance measurements and an analysis of the source code of these build systems, we find that the differences in task scheduling between build systems do not significantly affect performance.

1. Introduction

The performance of compilation is important. Developers rely on compilers to give them type-checking errors and intelligent warnings; and once their code is compiled, it can be tested. Increasing the speed of a compiler increases the speed at which developers get feedback on their code and leaves them waiting less. In contexts where many packages have to be built at once (e.g. for binary package distributions like Arch Linux), improving the performance of compilation can lead to significant speedups.

All but the smallest codebases rely on a *build system* to manage compilation for them. Rather than invoking compilers by hand, developers can let a build system do so on their behalf; it will automatically compile the necessary files using the appropriate compilation flags. Build systems also provide incremental and parallel compiles (only compiling files that change, and distributing work across available CPUs), resulting in significant speedups.

Within a build system, the work of compilation is broken down into many small compilation tasks. For example, a C codebase can be compiled by building each source file individually, then linking the resulting object files together to form an executable or shared library. Importantly, some of these tasks depend on each other, and can be visualized as a dependency graph. Every build system has a *scheduling procedure* which selects tasks to execute from this graph and assigns them to the available CPUs.

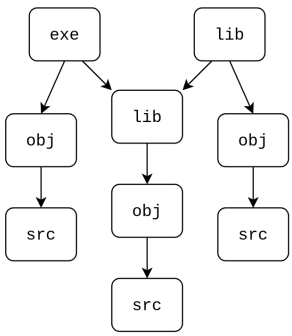

Figure 1: An example of a simplified dependency graph for a C codebase.

There is a vast academic literature studying task scheduling. However, commonly used build systems like Make[1] and Ninja[2] have not received academic scrutiny in this regard. If these build systems' ability to speed up compilation can be improved further, this would greatly benefit developer experience and resource efficiency. But there is a knowledge gap regarding the task scheduling within these build systems from an academic perspective.

In this paper, we examine the following research question: **what are the similarities and differences in the task schedulers used by build systems for compiling C codebases?** Empirically, we compare and contrast these build systems in terms of 1) their runtime performance and 2) their ability to maximize CPU utilization. Qualitatively, we look for explanations of these results in the task scheduling algorithms used by these build systems. By addressing this knowledge gap, we attempt to introduce an academic discussion of the application of scheduling to compiler design.

This paper includes both qualitative and quantitative approaches to the problem. Section 2 introduces the specific compilers and build systems that are examined by this paper, along with a formalization of the scheduling problem they have to solve. Section 3 describes the quantitative approach taken to profiling the build systems in question. Section 5 lays out the results of this profiling. Section 6 analyzes these results and tries to explain them through a study of the source code of these build systems. Finally, Section 7 enumerates the key takeaways from this analysis.

2. Background

This section describes the process of compilation for common C codebases, the compilers and build systems used to perform this task, and a formalization of the scheduling problem these build systems have to solve.

2.1. Compiling C Code

Non-trivial C codebases have existed for decades. In order to compile C code, each C source file must be individually compiled into an object file, and these object files then need to be archived or statically linked into a single static or dynamic library file. Furthermore, the compilation commands needed to build each source file may require a number of configuration flags (e.g. to specify the optimization level, degree of compiler warnings, and the characteristics of the machine being compiled for). If a single source file is changed, tracking which files must be updated is difficult to do manually, especially for larger projects with several levels of nesting.

The Make build system was introduced in 1975 to simplify the process of building C codebases [1]. It allowed developers to store their compilation commands in a central build description, and it would execute those commands to only rebuild files that had become outdated. Furthermore, Make could execute multiple tasks in parallel: if two files were outdated but did not depend

[1] See https://www.gnu.org/software/make/.
[2] See https://ninja-build.org/.

on each other, their compilation commands could be executed simultaneously. While there are multiple implementations of Make available, GNU Make[3] is one of the most common.

The Ninja build system[4] began development in 2010, as an alternative to Make. Unlike Make, Ninja's build descriptions are not meant to be written by hand; instead, users are expected to use a separate tool that will generate a build description for them, such as CMake. This meant that Ninja's build description format is much simpler, and the entire codebase is significantly smaller than e.g. GNU Make. Still, Ninja does the same work of finding files to rebuild and executing tasks in parallel.

The Tup build system[5] began development in 2008, with a distinct focus on minimizing rebuilds by detecting implicit dependencies. Tup executes compilation commands within an emulated file system: when a C compiler tries to read an included header file, Tup will detect and remember this implicit dependency, and will automatically recompile the relevant source file when the header file changes [2]. Tup is also fully capable of parallelization.

The reference implementation of the Zig language[6] is capable of compiling C code and has its own build system (configured using Zig itself). The Zig compiler uses LLVM and so benefits from its repertoire of optimizations and tooling. Since the Zig Build System is built into the implementation, its C compiler is inherently parallel: it will automatically make use of all available CPUs to compile a given set of source files. This means that Zig can be used with or without external build systems like Make, Ninja, and Tup. Since it performs the same work in either case, it provides a uniform environment to benchmark these build systems in.

2.2. The Scheduling Problem

Given a build description, which enumerates the artifacts to generate and the procedures necessary to generate them, a build system constructs a *dependency graph* of the tasks to be performed and how they relate to one another. In a dependency graph, nodes are tasks (such as the compilation of a particular C file) and edges are dependencies (if a has an edge to b, then a depends on b and can only be executed once b is complete). Nodes are associated with costs, measuring the runtime of each task.

The granularity of nodes, and thus the size of the graph, varies between build systems. While Make, Ninja, and Tup are generic tools that can only operate on whole files at a time, the Zig build system has domain-specific knowledge about its tasks and can break down compilation steps per C function rather than per C source file. Systems with more granular tasks have more opportunities for parallelism and so are expected to more efficiently schedule tasks.

Knowledge about the (approximate) runtime of each task can aid schedulers, allowing them to prioritize executing chains of dependent tasks and to predict when CPUs will finish executing each of their tasks. Once again, Make, Ninja, and Tup have no domain-specific knowledge about their tasks, and so are unable to predict their runtimes. The Zig build system could theoretically predict the runtime of each of their tasks, using heuristics on the size of each function or object being compiled, but this information becomes available during compilation rather than statically.

The general scheduling problem being answered here is thus: given a static compilation graph, with partial estimates of the runtime of each compilation step, and N identical processors to schedule compilation tasks to, what is the optimal assignment of compilation steps to processors, in order to minimize the time until every task has been executed?

Because the runtime of each compilation task is not fully predictable, and because the compilation graph may not be available entirely ahead of time, there is no single optimal schedule, and perfect optimality is unachievable. This is a dynamic scheduling problem: scheduling decisions may be made during the course of the build process, depending on the runtime of executed compilation tasks.

In the literature, Y.-K. Kwok and I. Ahmad [3] consider many different static scheduling algorithms for these kinds of dependency graphs. While they often consider these graphs in terms of precedence constraints ("task A must run after task B"), we consider them in terms of nodes and edges. In particular, we consider dynamic non-preemptive scheduling for non-parallel tasks in a DAG that does not contain conditional branches. Tasks have unknown computational costs, negligible communication time, without duplication. Tasks are scheduled to a limited number of fully-connected processors. There are a number of scheduling algorithms for this case enumerated by their survey, these are not found in the build system implementations we explored.

3. Methodology

We measured the performance characteristics of these build systems by using them to compile some C/C++ codebases. The methodology for the experiments performed is detailed in this section. Firstly, we found a collection of C/C++ codebases that are representative of the general case and that can be processed uniformly by automatic tools. As these codebases used a variety of build systems, we designed a system for extracting relevant C/C++ compilation commands from an arbitrary build configuration. We used it to generate functionally identical build descriptions for Make, Ninja, Tup, and Zig. Finally, we benchmarked the runtime and CPU utilization of these build systems against these packages, and tested the hypotheses that Zig's build system was faster than each of the others.

In order to perform benchmarking uniformly and reliably on a large number of codebases, an automated process was required. But the lack of universally preferred build systems for C/C++ meant that any repre-

[3] See https://www.gnu.org/software/make/.
[4] See https://ninja-build.org/.
[5] See https://gittup.org/tup/.
[6] See https://ziglang.org/.

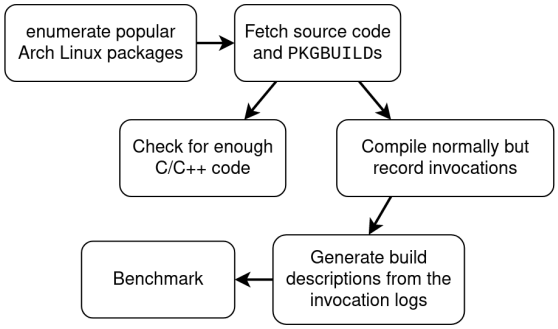

Figure 2: The process of empirically comparing build systems for this paper.

sentative sample would contain a variety of build systems and each codebase would require human interaction to build. We sidestepped this problem using the Arch Build System[7] (also known as ABS). ABS is a collection of tools and conventions that allow developers to describe the build process for any codebase in a simple, uniform way. The Arch Linux distribution uses ABS to automate the build process for the tens of thousands of official and unofficial packages available to users. Every Arch Linux package has an ABS configuration, and the distribution's popularity means that an Arch Linux package for any popular application will be available. We decided to use commonly-installed Arch Linux packages as our source of C/C++ codebases to benchmark.

While the use of Arch Linux packages simplifies later steps in the experiment, there are too many to compile and benchmark practically; in fact, it would be wasteful of resources to do so. We ranked Arch Linux packages based on their popularity as measured by a community tool called `pkgstats`. The `pkgstats` project attempts to rank Arch Linux packages based on how many systems they are installed on (out of the roughly 16,000 systems running `pkgstats`) [4]. Packages that are installed on more systems are by some measure more important; improving their compilation process, which is the purpose of this thesis, would be more effective in overall resource efficiency than improving the compilation process of less popular packages. We queried the `pkgstats` database using its REST API and examined the 500 most popular packages it listed. While some of these are not C/C++ codebases, there were enough to for the purpose of these experiments.

We now had a selection of codebases which could be built using ABS. However, each codebase is free to use any build system possible; the ABS configuration simply indicates how to execute the codebase's build system. Furthermore, the build steps configured for ABS may include additional procedures that are not relevant to this thesis, such as the generation of HTML documentation. These considerations make the codebases' existing build descriptions unsuitable for direct use in benchmarking. Instead, we developed a system called `wizardry` which watches an arbitrary build system compile a package, records the C/C++ compiler invocations that system used, and reformats them into a much simpler build description for any other build system. Because only C/C++ compilations are recorded, any build steps not relevant to this thesis are implicitly omitted.

[7]See https://wiki.archlinux.org/title/Arch_build_system.

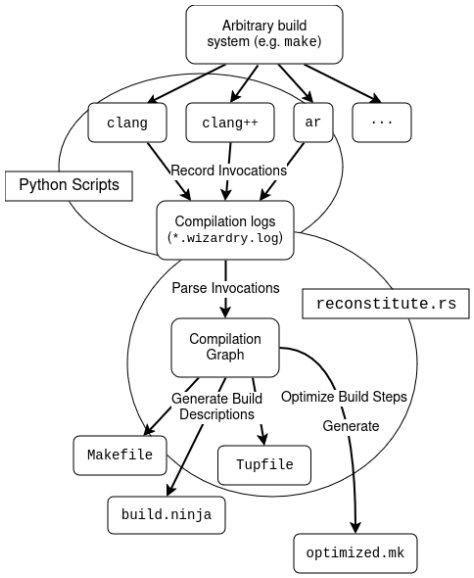

Figure 3: An overview of the `wizardry` system, used to extract relevant build steps from any build system and reformat them into an arbitrary build description.

The `wizardry` system works by taking advantage of the `PATH` environment variable on Unix systems. We wrote simple Python scripts to emulate `clang`, `clang++`, and `ar` (which generates static libraries), and placed these in a directory that was then prepended to the `PATH` variable. Any build system using this modified `PATH` variable which attempts to use Clang will invoke our scripts instead. These scripts simply record the command-line parameters they receive, which are intended for the respective compiler tool being replaced, then calls the underlying compiler tool without any modification. Since they act exactly like the underlying compiler tool, at least from the perspective of any unsuspecting users, they do not interfere with the build system used by the package being inspected, and are highly reliable. These scripts write their invocation information to a log file in a simple binary format to ensure that parameters containing spaces or other punctuation are not misinterpreted.

The primary component of `wizardry` is a Rust program called `reconstitute` that processes recorded compiler invocation logs and generates build descriptions. It addresses three major concerns in the experimental procedure for this thesis: 1) the normalization of compiler invocations, 2) the optimization of compilation graphs, and 3) the generation of build descriptions, all in a thousand lines of Rust code. It relies heavily on existing infrastructure that has faced these problems before.

C/C++ compilers are difficult to model properly because they are incredibly flexible tools. For example, the GCC compiler accepts more than two thousand options, which cover the C/C++ language features enabled, the optimizations attempted by the compiler, architecture-specific code generation switches, and even the logging of header file dependencies for incremental compilation in build systems. While these switches can affect the properties of the produced code, they are ir-

relevant for the purpose of this thesis: they do not have any bearing on the scheduling properties of the build systems used. In fact, they make it more difficult to test the hypotheses of this paper: a single instance of the Zig build system can only consume C/C++ sources with compatible compiler configurations, so incompatible configurations in the same project prevent the build system from scheduling multiple tasks. It is important to normalize the compiler flags consumed so that irrelevant flags are removed and the remaining flags have predictable behaviour that is easy to model.

We make use of parts of the Zig and Clang compiler infrastructures in order to normalize the command-line parameters parsed by `reconstitute`. The goal of normalization is to produce a minimal configuration that will compile without failures. Of the massive number of available options in these compilers, very few are actually relevant: they are listed below in Table 1. In a sample invocation like `clang -Ibar -target native -o foo foo.c`, knowing that `-target` (an irrelevant option) consumes the following parameter is necessary in order to parse the rest of the command-line. The Zig compiler stores this parsing information in a large auto-generated array in its source code (which it derives from a similar table in the Clang source code); we extracted it by adding a few lines of Zig code to traverse the array and print each option. For each category of option parsing, a plain text file containing the names of those options is stored; these are loaded into the `reconstitute` program, which replicates some of the parsing logic from the Zig compiler.

Flags	Purpose
`-I` and related	Include directories
`-D`, `-U`	Macro definitions
`-E`, `-S`, `-c`, `-shared`	Output type
`-std`	Language standard
`-L`, `-l`	Linking to libraries
`-o`	Output file name

Table 1: A complete list of the compilation flags retained by the `reconstitute` tool when it parses compilation logs.

In order to generate a build description that uses the Zig build system, any parsed set of invocations have to be optimized. Build systems for C/C++ universally operate by first generating an object file for each source file (in parallel), and then executing a linking step which combines the object files. The Zig C frontend can perform both portions of work with a single invocation, using its own scheduler rather than that of the outer build system. However, source files using incompatible flags (e.g. macro definitions) cannot be compiled in the same Zig invocation. Optimally picking invocations to merge like this quickly becomes intractable; to produce the best possible results, we use `egg`, a super-optimizer that can efficiently represent many combinations of equivalent nodes [5]. Once the invocations are parsed, we attempt to merge random invocations together and store the results in `egg`. After a large number of merges, we extract the best available combination of invocations as the final optimized build steps.

Once a compilation database representing a codebase is constructed, it can be translated into a build description for any build system. Simple functions for formatting invocations in the Make, Ninja, and Tup formats was implemented as the last step in `reconstitute`. For optimized build descriptions taking advantage of Zig's scheduler, more than one invocation may exist; they are executed by Make, so the same generation code is reused. At this step, a list of the artifacts generated by the regular and optimized build descriptions are also output; these are used to prevent incremental recompilation from occurring, as it is not relevant to the topic of this thesis.

Finally, we executed the generated build descriptions and benchmarked the build systems running them. We used the Linux `perf` tool[8] to collect information directly from the OS scheduler about which tasks were allocated to which CPUs. We began by collecting the total runtime of each build. is identical: they use the same compiler (the Zig C frontend) to compile the same codebase to the same set of artifacts. We control for variations in disk caching by running each build system five times and discarding the results from the first build. Any significant difference in the runtime of the build must be attributed to the build system itself, allowing us to compare their overheads. In order to judge the degree of parallelism used by each compilation, we measured 1) the CPU resources consumed by the build in total, 2) the CPU resources that were available to the build over its runtime, and 3) the CPU resources consumed by other processes on the system (i.e. system overhead). These allow us to infer the degree of parallelism of the build: if the build system perfectly scheduled tasks to CPUs, it would consume all of the CPU resources available to it; if some CPUs were under-utilized, that idle time is accounted for. We minimized the system overhead by stopping unrelated processes and shutting down unnecessary subsystems like networking and graphics, but the measured system overhead represents potential for better parallelism that was not made available to the builds.

We tested six hypotheses using the benchmarking data collected: that the Zig build system is on average faster (has a faster runtime) than every other build system, and that the Zig build system schedules processes more effectively (has less idle time relative to total runtime) than every other build system on average. We averaged the four runs of each build system on each codebase, paired the values from Zig and another build system for each codebase, and then performed a one-tailed Student's t-test to determine the probability of the null hypothesis that Zig had an equal or greater runtime or that Zig had an equal or lesser CPU utilization (relative to total runtime). Before performing these calculations, we decided a significance level of $p = 0.05$ – if the null hypothesis is true with this probability or lesser, the alternative hypothesis will be assumed.

[8] See https://perf.wiki.kernel.org/index.php/Main_Page.

4. Responsible Research

The correctness, reproducibility, and ethical implications of any research are important factors to consider. In the case of this thesis, correctness depends on the validity of the data set as a representative sample and of the methodology as a means of answering the stated hypotheses. The reproducibility of this work depends on the completeness of the description of the method used. There are relatively minor ethical implications to this work, but they revolve around the expenditure of energy in terms of electricity for the experiments.

4.1. Correctness

The results of this paper are intended to be representative of C/C++ codebases from the perspective of developers. To this end, we selected codebases that many users have installed on their systems (at least on Arch Linux). We assume that developers often interact with codebases that have many users, since this results in more bug reports, feature requests, and more interactions from the open-source community in terms of code contributions.

Due to limitations of the methodology, we could not process all of the codebases in our original data set. Some codebases use particular compilation flags that the `reconstitute` tool could not process correctly, while others could not be inspected as they mandated the use of the GCC compiler instead of our wrapper scripts around Clang. It would be unreasonable to spend a significant portion of time implementing support for this small number of packages, when the vast majority are successfully processed through the `wizardry` system.

The benchmarking performed as part of the experiments here did not use incremental compilation. This means that they always compiled every source file in the project from scratch. This is not wholly representative of the normal developer experience: developers are usually working on a few projects at a time, and once they have downloaded and built them once, incremental compiles drastically reduce the effort necessary to recompile them even as code is modified. However, the performance of full compilation is still important for developers, as it is used in continuous integration (CI) builds. We focused on full compiles because they are significantly easier to benchmark and are more sensitive to the scheduling algorithm used.

4.2. Reproducibility

We designed the experimental procedures for this thesis to minimize manual intervention, allowing them to be executed uniformly by anyone. The source code for the experiments is available at git.sr.ht/~bal-e/build-system-scheduling. The majority of the experiments is implemented using a single `Makefile`: it generates the list of packages to use, fetches their ABS bulid configurations and source code, compiles them and records compilation graphs, uses `wizardry` to generate new build descriptions, and performs benchmarking. A separate set of tools (also in the repository) are used for extracting information from the recorded performance profiles, testing the hypotheses against them, and producing the graphs in this paper. The steps for using these scripts to reproduce the experiment are detailed in a `README` file.

4.3. Ethics

The primary ethical consideration here is the use of energy in performing the experiments. While performing these experiments did spend (or waste) some CPU time that could have been spent performing more valuable processing, this is a nearly insignificant consideration given the power efficiency of modern CPUs. Furthermore, should the results from this paper result in developers switching to faster and more resource efficient build systems, the costs of running these experiments will be offset entirely.

5. Results

The following results were collected on a Framework Laptop 13, with a 4-core 2.4GHz 11th Gen Intel® Core™ i5-1135G7 running with 8 threads at 3.1GHz. The machine has 16 GB of DDR4 RAM and the experiments were performed on an external 500 GB Samsung PSSD T7 on BtrFS. It is running Arch Linux with the 6.8.10 upstream kernel (with manual configuration). Here is a breakdown of the data set sampled:

- The top 500 packages from `pkgstats` were selected.
- 477 unique "base" packages were found (some base packages result in multiple actual packages being built).
- 467 were successfully downloaded (some could not be prepared due to conflicting dependency packages).
- 396 packages contained between 2,000 and 300,000 lines of C/C++ (an upper limit was imposed to restrict the runtime of the experiment).
- 385 packages were successfully compiled and their compilation logs were extracted (some packages could not be built due to unlabeled build-time dependencies or compiler misconfigurations).
- 286 packages were processed by the `reconstitute` script to generate new build descriptions (some failed because of incomplete compilation logs, or because

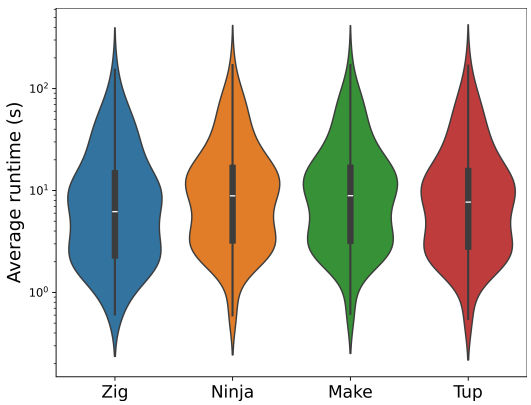

Figure 4: A logarithmic violin plot of the time it took each build system to compile, across the distribution of packages sampled.

optimizing the compilation graph led to out-of-memory errors).

- 114 packages were benchmarked with each candidate build system (originally, more than 200 were benchmarked; these results were discarded due to the Zig compiler automatically caching compilation data and misinforming compilation times, and new results could not be generated in time).

The Zig compiler's inherent build system is significantly faster than Ninja, Make, and Tup. Using Student's t-test with paired samples, the null hypotheses (that Make, Ninja, or Tup were individually as fast as or faster than Zig's build system) were rejected with $p < 0.05$.

System	μ	Med.	σ	t-stat.	p-value
Make	17.06	8.888	24.50	−5.234	$3.965 \cdot 10^{-7}$
Ninja	17.12	8.876	24.59	−5.254	$3.636 \cdot 10^{-7}$
Tup	15.20	7.698	22.68	−3.664	$1.909 \cdot 10^{-4}$
Zig	**13.52**	**6.020**	**20.30**		

Table 2: Statistics regarding the runtimes (in seconds) of the build systems across all packages. All values are rounded to four significant digits.

In addition to runtime information, we also collected basic CPU utilization data. From the moment the build starts, we consider any time the CPU spends outside the build system processes to be wasted. By aggregating this data and comparing it to the total amount of CPU time available to the build (which is the runtime of the build times the number of available CPUs), we can measure the degree to which a build system takes advantage of the available CPUs. In particular, we test the hypotheses that Zig's build system has a greater CPU utilization than Make, Ninja, and Zig.

System	μ	Med.	σ	t-stat.	p-value
Make	0.7174	0.8052	0.2539	−5.264	0.999
Ninja	0.7160	0.8087	0.2531	−5.303	0.999
Tup	0.6893	0.7782	0.2500	−3.457	0.999
Zig	**0.6698**	**0.7637**	**0.2599**		

Table 3: Statistics regarding the CPU utilizations (as ratios between measured CPU usage and the maximum possible) of the build systems across all packages. All values are rounded to four significant digits.

Surprisingly, Zig consistently had a lower average CPU utilization than any other build system: the null hypothesis was not rejected in any case. This means that it was not making better use of the available CPUs – but since its runtime was lower on average, it is doing less work and distributing that work across CPUs less effectively. The same is true to a lesser degree of Tup. This is more visibly demonstrated by Figure 5.

To gain further insight into the reason for this, we analyzed the amount of CPU time spent within each process during compilation. Time that could be attributed to a particular task, or to CPU idling, was averaged across the packages, per build system, and is presented in Figure 6.

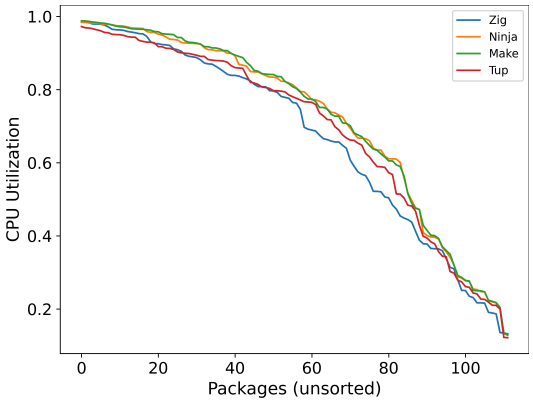

Figure 5: A plot of the degree of CPU utilization by each build system for each package, sorted (per build system) from highest to lowest.

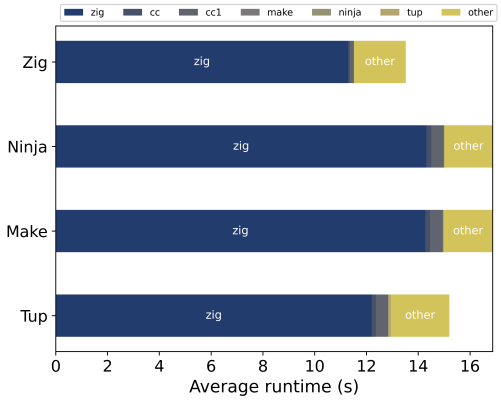

Figure 6: A breakdown of the average amount of time spent in each kind of process, or in CPU idling, during compilation for each build system. "other" refers to a small amount of system noise, or near-insignificant components such as `sh`.

In addition to the runtime and CPU utilization of build systems, information was obtained about the build experience for a large number of packages. We analyzed the runtime of building a large number of packages but we also analyzed the actual compilation invocations within those packages. We test the hypotheses that the average runtime for building a package with the Make build system is correlated with the number of source files or the number of lines of C/C++ code within it. In accordance with Table 4, we reject the null hypothesis that build runtime is not correlated with the number of source files; however, we cannot reject the null hypothesis that build runtime is not correlated with the number of lines of C/C++ code. The number of source files in a codebase more significantly affects its build runtime than the number of lines of code within it; indeed, in Figure 7 we see a number of codebases with few files but many lines of code that compile faster than other codebases with many files but few lines of code.

In conclusion, our results show that the Zig build system is significantly faster than Make, Ninja, or Tup, when used with the Zig C frontend to build common C/C++ codebases. The following section examines why this may be the case, and how these build systems' internals compare and contrast. Furthermore, we find a distinct lack of correlation between the number of lines

Measure	ρ	p-value
Number of source files	0.6090	$1.046 \cdot 10^{-12}$
Number of lines of code	−0.1370	0.1499

Table 4: Pearson correlation coefficients and p-values for estimating the runtime for a codebase using the Make build system.

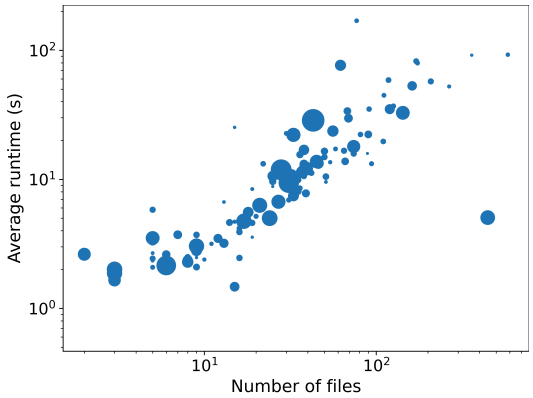

Figure 7: A log-log plot depicting how the runtime for building a codebase with Make varies along the number of source files (along the X-axis) and the number of lines of C/C++ code (the size of the points).

of code in a codebase and its expected runtime with the Make build system.

6. Discussion

In this section, we search for explanations for the results we have found: that Zig's build system is faster than others and that it does not have a greater CPU utilization than other systems. We attempt to explain these properties in terms of the scheduling algorithms underlying these build systems.

We analyzed the source code for the Zig compiler and determined that its task scheduler does not take advantage of the domain-specific knowledge available to it. Within the compiler [6], we found that C compilation commands were added in the order they were specified on the command line to a thread pool (by the `performAllTheWork()` function in `src/Compilation.zig`). The thread pool is a simple first-come first-serve task scheduler, part of the Zig standard library (at `lib/Thread/Pool.zig`); it maintains a global queue of pending tasks, protected by a mutex, and a set of worker threads that poll it for new tasks by popping it from the front. The compilation tasks are specified at the granularity of whole source files to compile. The individual compilation tasks are implemented by `updateCObject()` in `Compilation.zig`: they simply spawn new instances of the Zig compiler which will use the raw Clang frontend contained within it via LLVM. As such, Zig operates very similarly to the other build systems it is being compared to: but it hides these characteristics as implementation details.

The GNU Make build system uses a depth-first traversal of the dependency graph in order to select tasks to execute. While this characteristic is relatively well-known, we examined the source code of GNU Make 4.4 [7] to confirm this claim. In the file `src/remake.c`, a high-level `update_file()` method is responsible for executing a particular rule in a `Makefile` (if necessary); it calls an internal `update_file_1()`, which checks the dependencies of the rule using `check_dep()`. This will recursively call `update_file()` to ensure the dependency is prepared. Because `update_file_1()` traverses dependencies in the order they were specified in the `Makefile`, GNU Make is performing a depth-first post-order search through the dependency graph. The actual update work is performed by the commands specified in the `Makefile`; Make will spawn the necessary processes and continue, unless a user-specified limit is reached or if all remaining jobs are waiting for processes.

The Ninja build system prioritizes tasks based on critical dependency chains. In Ninja's source code repository [8], we found the `Plan::computeCriticalPath()` function of `src/build.cc`. It assumes, as a simple heuristic, that every build step will take the same amount of time to execute. For each build step, it then determines the minimum number of steps that must follow after it for the originally requested artifact(s) to be generated. This is the length of the critical path for that step. Having computed this information, Ninja then performs earliest deadline first scheduling: in order to generate the top-level artifact as soon as possible, it will prioritize build steps that have to occur the longest time before that overall deadline. Of all the build systems examined, Ninja performs the most thorough analysis of the dependency graph, and uses the most intelligent scheduler possible. However, our results show that Make is still faster than Ninja.

While Tup performs very interesting analyses, its source code proved very difficult to traverse. Unlike Zig, Make, and Ninja, Tup is not very widely used, and has not benefited from many external contributions over the years. As such, the code is difficult to navigate and documentation is sparse. While we found some code for managing parallelism, we were unable to find basic steps such as `fork()`/`exec()` pairs for spawning new processes. The technical documentation for Tup, such as [2], does not adequately describe Tup's scheduling mechanism, leaving it as an additional but relatively unhelpful data point in this thesis.

We are forced to conclude that Zig has a significantly smaller runtime than any other build system because of its own overhead, not because of any scheduling algorithm. If Make and Ninja introduced significant overhead to the build process, then their share of CPU time consumption would be visible in Figure 6; instead, Zig spends more time performing the same work just because it was invoked from an external build system. The only explanation for this is that Zig has a significant overhead; every time it is invoked, it spends a large amount of time in initialization or teardown procedures.

7. Conclusion

We examined task schedulers within build systems used to compile C codebases, and we concluded that they do not have a significant impact on the speed of compila-

tion. However, we still have important takeaways from this experiment:

1. There is a significant overhead to individual compiler invocations. While changes to the scheduling algorithm may not improve build system performance, better integration with the tools used for compilation yield massive speedups.
2. For this reason, the Zig Build System is a good choice for C/C++ developers looking to improve their compile times. By compiling their code faster, they will be able to get feedback on their code more quickly, and for even larger codebases, this will measurably improve resource consumption.
3. While C/C++ build systems have thousands of configuration options, most of them are not strictly necessary: the vast majority of packages can be compiled successfully using only the ten most important flags.
4. The methodology of this thesis is a major contribution. It can be used to experiment with a large number of real-world codebases in an automated manner for any stage of the compilation process. For example, this could be used to benchmark the performance differences between two linker implementations in practice.
5. Using the `reconstitute` tool, it is now possible to translate between build system descriptions arbitrarily, and even to optimize them to minimize the number of compiler invocations (assuming Zig-like functionality for compiling multiple source and object files into a single object file). This can be used to experiment with new build systems or potentially to help transition a project between build systems.

Future work in this topic should consider the benefits of breaking down tasks into smaller components: this provides more opportunities for parallelism and more predictable runtimes for each component. This may have to be implemented first in a practical compiler implementation before it can be analyzed in an academic setting.

Bibliography

[1] S. I. Feldman, "Make – A Program for Maintaining Computer Programs," *Software – Practice and Experience*, vol. 9, 1979, [Online]. Available: https://doi.org/10.1002/spe.4380090402

[2] M. Shal, "Build System Rules and Algorithms," 2009. [Online]. Available: https://gittup.org/tup/build_system_rules_and_algorithms.pdf

[3] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, pp. 406–471, 1999.

[4] P. Schmitz, "Arch Linux package statistics." [Online]. Available: https://pkgstats.archlinux.de/

[5] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and Extensible Equality Saturation," *Proc. ACM Program. Lang.*, vol. 5, Jan. 2021, [Online]. Available: https://doi.org/10.1145/3434304

[6] A. Kelley, "Zig: A general-purpose programming language and toolchain for maintaining robust, optimal, and reusable software.." Jun. 2024. [Online]. Available: https://github.com/ziglang/zig

[7] "GNU Make." Jun. 2024. [Online]. Available: https://savannah.gnu.org/projects/make/

[8] E. Martin, "Ninja: a small build system with a focus on speed." Jun. 2024. [Online]. Available: https://github.com/ninja-build/ninja