Lessons and Pitfalls in Building Firefox with Tup

Guillaume Maudoux and Kim Mens
UCLouvain (Belgium)
Email: {guillaume.maudoux,kim.mens}@uclouvain.be

***Abstract*—Build system implementations are surprisingly numerous for the single common purpose of assembling software. With this variety, picking the right one is a complex task. And even more difficult is the migration to a new build system, with uncertain benefits at the end. Software maintainers and release engineers need better comparisons of build systems and precise catagorisation on which to base an informed decision. As a first step toward that goal, we experimented building Firefox with Tup in replacement of Make. We report here our experience at migrating and comparing the build systems. We also describe interesting features of Tup and we discus Mozilla's Firefox usage as a benchmark for build systems.**

I. Introduction

We started investigating Mozilla's build system a year ago, in the hope of finding a large code base for testing different build systems in a realistic setup. We settled on Mozilla's Firefox because their build infrastructure is designed in such a way that different build systems can be plugged in. This is no coincidence, but an ongoing work at Mozilla to update their build system. The rationale being that picking the right replacement for Make would require testing different alternatives, and each alternative would require a complete port. With a generic build definition, they can test different build systems and let experimentations drive the selection of the next build system. Other considerations, like gaining control over the build definition, and using a widely known language to do so were taken into account when discussing this change [1, 2].

We started implementing the Tup backend last in August 2017, when Mozilla's support for it was minimal. It later appeared that Tup was also the next target build system for Mozilla itself, which led to two independent implementations (ours and theirs). Mozilla's effort to use Tup has focused on producing reusable, clean code by fixing one issue at a time. Our focus was on getting the build to work, regardless of the code quality. Over the time, both implementations have converged and as of August 2018, Mozilla's implementation should be preferred as it is now complete and is the only one that will be further maintained and updated.

This paper presents the insights we gained on three main areas. With this work, we investigated and gained an in-depth knowledge of Tup's capabilities and interals. We also learned a lot about Firefox's build system design and their tactics to tame their complex codebase. Finally, we accumulated experience in migrating from one build system to an other, and learned the hard way that the process is theoretically simple but very technical in practice.

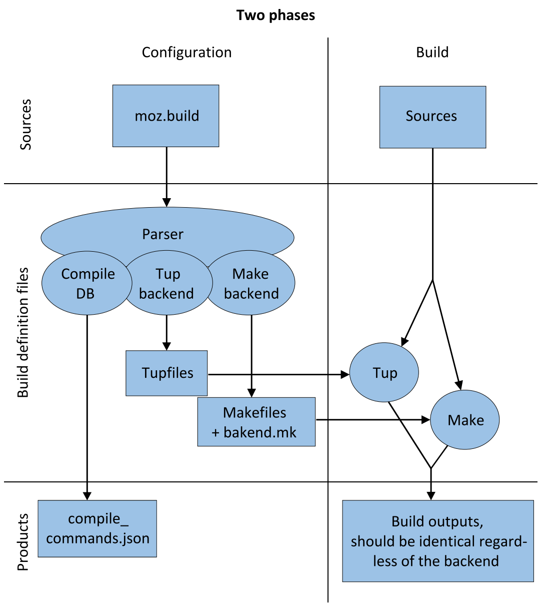

Fig. 1. Firefox's build system is split in two phases and can use different backends to perform builds. Depending on the intended backend, different build definitions are generated.

II. Context

For readers who may not be already familiar with these, we briefly depict Mozilla's build system, Firefox itself and the two build systems considered this article: Make and Tup.

a) Mozilla's build system: Mozilla's modular build setup works in two phases, as depicted in Figure 1. moz.build files describe the build with a Mozilla-specific python DSL. This format is parsed and Makefiles are generated in such a way that the remaining of the build can be handled by Make. As stated before, this design accepts new emitters, or back-ends to be used in place of the Makefile generator. This is how the Tup backend was implemented. Make and Tup are examples of backends used for compilation, but the same mechanism is also used to extract information from the build system to be used by other tools. For example, a backend generates a compilation database for advanced autocompletion in some IDEs [3, 4]. This flexibility makes it possible to test different build systems and inspect Mozilla's Firefox compilation.

b) Firefox codebase: Firefox is a libre, open-source browser with a large code base. The core application contained about 4M lines of code in 2013, and was growing steadily [5]. This does not count tests and other configuration files. At that time, the full repository contained about 15M lines of code, and has now reached above 36M lines of code as of 2018 [6], which makes it bigger than the Linux kernel and Libre Office according to Open Hub. It is written in a variety of languages. C++ forms the core application, supplemented by JavaScript on top of the core engine. The repository contains also HTML, C, python, java and has recently seen the introduction of Rust, Mozilla's designed systems programming language. A large code base, with a variety of languages forming an open-source and well known application. All these features makes it a good candidate for meaningful build systems comparison.

c) Make: Speaking of build systems, Make is the reference application in that domain. It was designed at Bell Labs in 1976 and was derived and reimplemented many times since then [7, 8]. Make's configuration language used in Makefiles is well known by a large proportion programmers. Over the years, Make has seen many disussions about its shortcomings, and even more discussions about good practices and usage. Most notably, the paper "Recursive Make Considered Harmful [9]" discussed the good usage of Make on large software projects, advocating for efficiency at the expense of modularity. Strinking the right balance between conflicting goals is still a challenge for current system implementations [10]. Make has always been the build system of Firefox and is tightly integrated in its code base.

d) Tup: From Tup's home page, we see that "Tup is a file-based build system for Linux, OSX, and Windows. It inputs a list of file changes and a directed acyclic graph (DAG), then processes the DAG to execute the appropriate commands required to update dependent files. Updates are performed with very little overhead since Tup implements powerful build algorithms to avoid doing unnecessary work. This means you can stay focused on your project rather than on your build system" [11]. Besides optimization and careful implementation, what makes Tup truly special is its ability to trace command executions to obtain real dependencies and outputs. Validating the declared dependencies with the observed file accesses adds a safeguard against imprecise declaration and potential speedups that we will discuss later.

III. Research objectives

From the above description of Tup, it is clear that Tup is focussed on performance of updates. Detailed explanations on "avoid doing unnecessary work" and "stay focussed on your project" are to be found later on Tup's home page or in [12]. In a nutshell, avoiding redundant work is the aim of a minimal incremental build system. Minimalism ensures that up-to-date outputs are maintained without running the tasks that produce them. Tup claims that it is minimal in that sense. "Stay focused" hides the even more complex idea of correctness. Some build system can get corrupted, or out of sync and it is not uncommon for developers to clean their build tree and start again from scratch. A correct build system is one that always rebuilds what needs to be rebuilt. When an incremental build is complete, the binaries reflect the sources exactly, and the developer need not worry about the build system's state and shortcomings. She therefore can "stay focused on her project".

To summarize, Tup has three claims, each of which we will try to verify in this paper.

1) Performance of execution,
2) Minimal incremental updates,
3) Always correct build outputs.

These features arise from the design and careful implematation of the algorithms detailed in [12]. They are hard to demonstrate by experiment as they should hold by design. We have nonetheless tested Tup's efficiency at building Firefox and gathered the results presented in the next section.

Minimal updates and correctness can be investigated by a deep understanding of how Tup works and have real impacts on Tup usage, especially on large software systems like Firefox. It must first be noted that minimal updates and correctness somehow conflict. Stricter notions of correctness will require more rebuilds. While still being minimal, the build system will rebuild much more than with other build systems, leading to the impression that it is uselessly recompiling more parts. This is exactly what happens with Tup.

IV. Building Firefox with Tup

Tup focuses on correctness and speed, at the expense of expressivity or usability whenever they conflict with the former aspects. In some sense, Tup is minimalist and tries to do one thing well: building. The configuration language of Tup is not very expressive. It allows to specify commands, inputs and outputs with basic support for variables and conditions. To be fair, Lua integration is provided for more advanced features but was not tested in this article.

We will report here first on our experience with Tup, separated in two key aspects: correctness and speed. Expressivity will be discussed later on, in Section VI.

A. Correctness

Correctness is a key feature of build systems. It can be defined roughly as the property of producing valid products despite optimizations like incremental compilation [10, 13]. In Tup, correctness is achieved by maintaining the state of the build in a separate database and by tracing all the file accesses performed by build commands.

Maintaining the state of the build in an external database, allows Tup to detect changes to the build description. Whenever a command is added, removed or modified, Tup will take proper actions to update the build when invoked [12]. By design, this is difficult or impossible to achieve with stateless build systems like Make. In particular, they cannot detect removed commands as there is no trace of them on the next invocation. This is not specific to Tup. Most (if not all) recent build system implementations now rely on persistent storage.

```
tup error: Unspecified output files - A command is writing to files that you didn't specify in
 ↪  the Tupfile. You should add them so tup knows what to expect.
tup error: Expected to write to file 'libxul.so' from cmd 15846 but didn't
tup error: Missing input dependency - a file was read from, and was not specified as an input
 ↪  link for the command. This is an issue because the file was created from another command,
 ↪  and without the input link the commands may execute out of order. You should add this file
 ↪  as an input, since it is possible this could randomly break in the future.
```

Fig. 2. Typical errors produced by Tup when inputs and outputs are not properly specified. These errors are reported respectively when a command 1) writes to a file that was not specified as an output, 2) does not write to a file specified as an output and 3) tries to read a file that is not declared as an input.

Unlike Make, Tup detects changes to the build commands, triggering the required command invocations.

By tracing file accesses of each and every command in the build plan (a.k.a. build steps), Tup ensures that dependencies and outputs are correctly specified. In particular, Tup will refuse to build a command that uses undeclared dependencies. Some typical error messages can be seen on Figure 2. While this allows to detect hidden dependencies and fix them, we often stumbled upon this constraint during our migration because it interrupts the build, and the Tupfiles need to be generated again. Tup does not allow to bypass these issues. Tracing is used as enforcement of the declared dependencies and for speed optimizations as discussed later. Tup does not dynamically detect and change the build plan from the collected information.

For example, we hit this constraint with fake static libraries (.a.desc files) built by Mozilla in place of the usual static libraries (.a files) whenever the static library will only be linked with other libraries, and not exported or otherwise directly used. In that case, the .a.desc file contains only a list of the objects that should be present in the archive. That list is then parsed and the real objects fed to the linker instead of the .a archive itself. This avoids the creation of real .a files that can be quite time and disk-space consuming.

This optimization conflicts with Tup's need to know all the dependencies of a command explicitly. Whenever a command links against the .a.desc, it implicitly depends on all the objects listed therein and this lists needs to be explicitly expanded for Tup when generating Tupfiles.

Tup strengthens the build description by detecting undeclared and hidden dependencies. They all must be declared in the Tupfiles or Tup will error out.

In Mozilla's implementation this is now avoided by skipping fake libraries. They are not generated, and linkers are fed directly the full list of objects. This is an optimization over our version. Since the full list of objects needs to be expanded during configuration anyway, there is no reason not to feed it directly to the linker.

In our first implementation, we avoided this issue by using Tup groups; a feature that allows to some extent to alleviate the need to list all the required dependencies by providing a rough super-set of these. By adding all the objects to the same Tup group linker invocations can depend solely on it, at the expense of introducing stages in the build, because linking will have to wait for all the objects to be compiled before running. The real dependencies are then obtained at execution by tracing the command. This ensures accurate detection of updated inputs despite the over-approximation in the Tupfiles, and therefore avoids pointless rebuilds of all the libraries when a single object changes.

Tup's "groups" feature allows over-approximating the dependencies of a build step.

The technical issue of specifying all the dependencies also arises in a other places such as with unified builds. The unified builds optimization works by aggregating several C/C++ source file together. The compiler only needs to parse the presumably large set of included headers once for the unified set, reducing compilation time on large rebuilds and on header changes [14]. This technique speeds up full rebuilds significantly at the expense of slower incremental builds for small changes. In this case, the C/C++ compiler receives a dummy unified input file that includes the original C/C++ source files. From the point of view of Tup, the command depends on the dummy unified source file, but also on all the source files included therein.

This situation highlighted for us a specificity of Tup: source files are not required to be specified as inputs. In fact, the set of input files is always an implicit input to all the rules. The rationale being that Tup is able to detect the sources that are accessed by a particular command and source files have no impact on command ordering. The execution order is only constrained by generated files, as consumers cannot run before producers. To relate this to groups, Tup behaves just as if all the non-generated files belonged to a "sources" group, automatically added as a dependency of all the commands. We experienced this style of never specyfying source files in rules, and this results in unusual, if not confusing, rules that have no declared inputs at all.

With Tup, there is no need to specify source files as inputs. All the commands implicitly depend on them.

Tup is also quite pedantic on the outputs that a command is allowed to produce. The command must write to all its declared outputs, and nowhere else. The issue occurred with Rust code. Due to a lack of integration at the moment, Rust libraries must be compiled in a single step, and therefore discard all outputs outside of the expected output library. This prevents Rust from doing caching of builds, or requires to build Rust libraries outside of the build and outside of Tup dependency tracking. Rust also suffixes libraries with a hash of their content. To work around these issues, we had to wrap Rust invocations with a cleanup script that removes extra files and renames the generated library.

Tup requires commands to produce the exact list of declared outputs, refusing commands that generate more, less or custom files on different invocations.

Even though these limitations sometimes prevent building with Tup, it is still possible to trick it in several ways. First, one can invoke some build steps before running Tup. The produced files are seen as plain source files by Tup. It however removes Tup's ability to detect changes and rebuild these files. In the absence of other change detection mechanisms, these files need to be rebuilt on each invocation. The other option is to build outside of the source tree, because Tup does not track files there (unless configured to). The required outputs can then be moved to the source tree once the command finishes. With this technique, complex commands can maintain a cache across builds. This is otherwise not possible with Tup constraints. The potential pitfall is that build correctness relies completely in the sub-command, as Tup has no knowledge of what happens outside of it's monitoring. Reaching that point means that Tup gets in the way more than it should. Commands such as Rust incovations require structural modifications to work properly with Tup. Mozilla's implementation was defered until they updated the Rust toolchain to better support Tup [15]. This was made possible because Rust already intended to support that usage [16], but may not be feasible with other projects or affordable to any Tup user.

Some tools are structurally conflicting with Tup. They have to be modified or Tup needs to be bypassed.

The complications induced by the strict enforcement of policies by Tup ultimately provide strong guarantees on the validity of the build results. While it is possible to inadvertently omit a dependency in Make, it is nearly imposible to do so in Tup. This makes building with Tup more robust to configuration errors or undetected interferences. In the long term, the balance leans in favor of strict build systems like Tup because migrating to Tup requires solving tricky issues only once, while an incorrect build system can produce subtle inconsistencies between the source code and the build products during any build invocation.

Tup's strong correctness guarantees come at the cost of strict constraints on invoked commands and on the build description. There is no free lunch.

B. Tup performance

Tup uses a specific data model to achieve high speeds with carefully crafted algorithms [12]. We first take a closer look at the three different aspects of Tup targeting performance, and then describe our benchmark on Firefox.

First, Tup comes with a monitor that can listen to filesystem events, collecting source changes on the fly. This optimization is nowadays used by all the build systems targetting large code bases, like Bazel, Gradle or Pants. The monitor removes the need to walk the source tree searching for changes on each build system invocation. This optimization is most noticeable for builds in an up-to-date workspace (a.k.a. null builds). In our tests, it reduced Tup execution time from one second to less than a millisecond. This result probably also applies to the build systems cited here-above.

When everything is up-to-date and Tup's monitor is enabled, Tup runs in less than a millisecond.

Also, Tup maintains the build graph in such a way that changes and rebuilds can be propagated upwards, without loading the whole dependency DAG [12]. Theoretically, this makes Tup very efficient at small rebuilds because the complexity of the algorithm depends mostly on the size of the update, and less on the size of the whole repository as is the case with Make.

Finally, Tup's build graph is enriched with exact dependencies detected by tracing the commands durig their execution. This ensures that build steps do not depend on files they do not need, and that there is no spurious rebuilds when such a file changes. That being said, tracing has also a performance penalty, and can reduce Tup's speed on filesystem-intensive tasks like linking Firefox's largest binary [17].

We ran a small-scale experiment to compare build speed with Make and Tup. We selected 47 consecutive pushes to Firefox[1] and compiled these changes incrementally in merge order. The pushes were taken from mozilla-inbound, the integration branch at Mozilla. A push is a set of commits that are added together, and tested as one atomic change by Mozilla's continuous integration system. The number of commits is limited to 47 because it is the largest range of consecutive commits that appeared to build correctly in the benchmarking enviroment. Building more commits would have required to use different build environments, or looking at more commit ranges. With 47 commits, we were able to measure the duration of 46 different incremental builds. Figure 3 shows the incremental build times of Make and Tup

[1]The full list of 47 pushes is accessible at https://hg.mozilla.org/integration/mozilla-inbound/pushloghtml?startID=102811&endID=102858

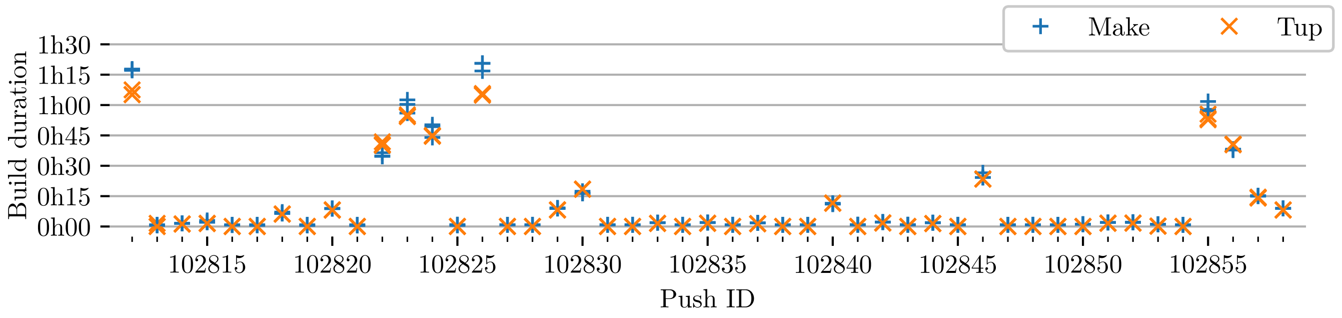

Fig. 3. Duration of three incremental builds for 47 successive changesets with both Make and Tup

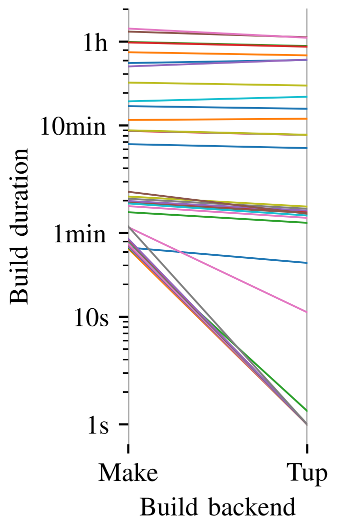

Fig. 4. Relative speed of Make and Tup at building incremental changesets. The build duration of each changeset with each tool has been averaged over the three runs. Durations were rounded up to the next second.

for each of these 47 commits. Three runs were made for each build system to give an idea of the variance. The first measure represents a full rebuild from clean sources, and shows how much time is needed for a full build of Mozilla's Firefox. With the default Make backend, a bit more than 1h 15m are needed to do so on our Intel Core i5-4310U which is a fairly recent 2 cores/4 threads CPU. All the builds were performed on the same machine, with the same SSD drive.

On the graph, we observe a lot of short builds, thanks to the incremental optimizations performed by Tup and Make. Also, the data series have the same shape, which shows that both backends compile about the same things. That being said, Make builds are generally slower than Tup ones on big rebuilds. Thanks to the three measures, we can affirm that it is no random artifact of the build environment. The most probable cause is that the Make build performs more steps than the Tup build. In some places, the Tup build avoids calls to wrappers and other fixtures used by Make. But it could also come from commands that are only described in Makefiles and not visible to the mozbuild frontend.

We get a more detailed view of Make and Tup's relative performance on Figure 4. Average build times with Make (on the left) are compared with average build times with Tup (on the right) for each of the 47 commits. The graph is in logarithmic vertical scale, so it is difficult to get precise values, but we see that the builds are indeed faster by a similar proportion for most large builds, with Tup being slightly faster.

Looking at smaller rebuilds, we see that they get faster with Tup, and sometimes much faster. This high speedup arises when Tup detects that there is nothing to do. In that case, Tup takes one second to search for modifications (and possibly even faster with its monitor running, as explained above), and then stops after deciding that no modifications means nothing to compile. Make on the other hand needs to recurse into all the subdirectories and build the whole dependency graph to detect the absence of anything to do. It further appears that Make executes some build steps on each invocation, making it impossible to measure the execution time of the Make process itself without the tasks being run. Unconditionally running tasks means that some were forced to run, most probably to avoid issues with incomplete dependency handling in Make. From that point of view, Make's lack of correctness guarantees translates into an execution overhead.

Tup is at least as good as Make on average, and much faster for small rebuilds.

V. Grasping Mozilla's build design

Another feature of Tup is that it allows easy extraction of the build graph. Tup contains in its database the whole build graph that was just built. It also provides a tool to draw the graph.

The complete graph is not easy to understand, as it contains thousands of nodes, one for each command and for each file. We took some steps to simplify the graph based on command labels and relative depth in the build tree to obtain the simplified version presented in figure 5.

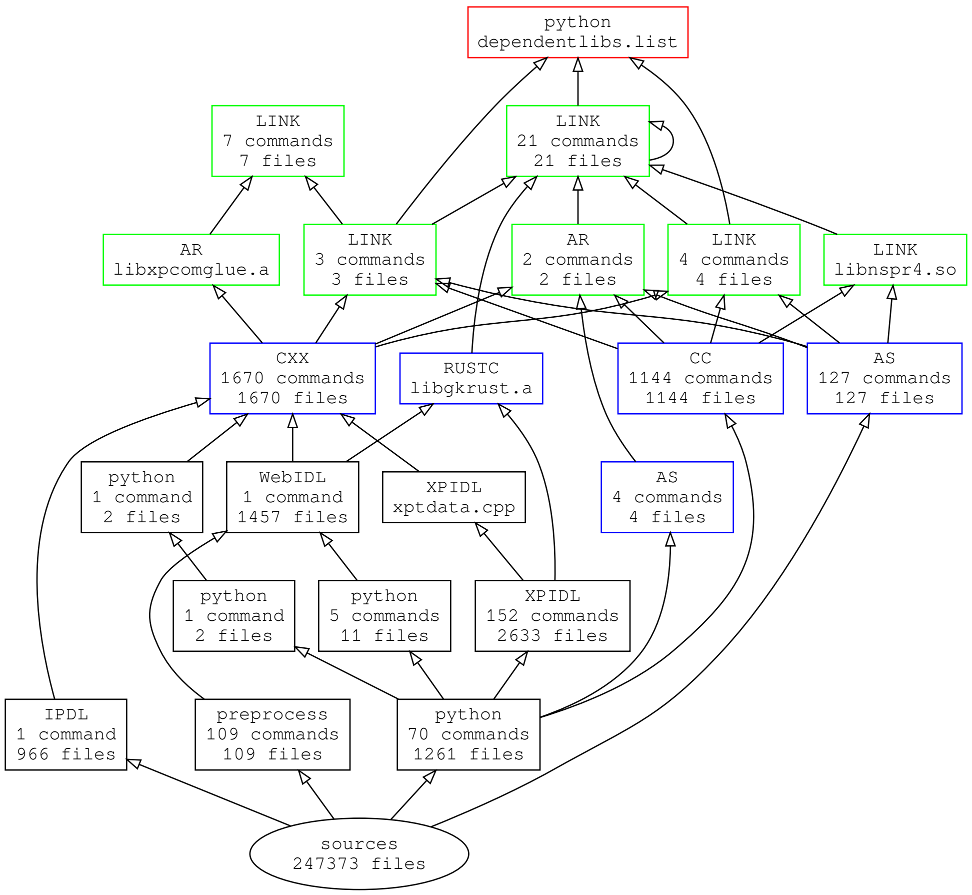

Fig. 5. A simplified view of Firefox dependency graph. The node grouping algorithm outlines different stages of preprocessing, compiling, linking and the final step of producing dependentlibs.list discussed in Section V. Nodes are grouped by type and by relative depth in the build system. Transitive dependencies have been removed to avoid edges clutter, hiding in particular the fact that all the nodes depend directly on the sources.

Tup stores the whole build plan (the build graph) in a database in an explicit form that can be queried easily.

It is interesting to discover that the build can be summarized in a handful of nodes, and divides itself naturally in stages. There is a preprocessing and generation stage where symlinks are created, where files are generated with python and other custom transformations happen. Afterwards, we observe the classical two-stage process of compiling (AS, CC, CXX) to object files, and then linking into programs, static libs and shared libs (LINK and AR). The ultimate stage is an oddity at Mozilla, where a dependentlibs.list is generated that lists all the libraries that libxul.so depends upon. This file is used to preload all these libraries at startup, possibly for performance reasons.

This graph was the best result obtained after several attempts. It groups commands by the maximal depth of their inputs. Files are merged with the command that produces them. This kind of transformation does not preserve the acyclicity of the graph, as a loop appears in the linking stage, where shared objects depend on other shared objects. The graphs could be improved through better projections and simplifications. We intend to work further on these aspects to improve on this work and generalize our results.

```
909M  gecko-dev/.tup     (Tup internal state,
              including and mostly due to Tup db)
2.2G  gecko-dev          (sources checkout only)
4.9G  gecko-dev/.git     (git repository)
5.8G  gecko-dev/obj-tup  (build outputs)
 14G  total
```

Fig. 6. Size of the main parts of our workspace after a complete build with Tup in a clean checkout.

Extracting that graph would *not* have been easy with recursive Makefiles. There have been attempts at extracting Makefile-based build system structure but they require to actually run Make in verbose debug mode and parse its output [18]. Tup provides a much more reliable way to access that graph.

Concerning that graph, it is worth noting that it requires a large amount of disk space. On our machine, after a fresh build of Firefox, we observed the memory usage described in

```
EXPORTS += [
    'MP3Decoder.h',
    'MP3Demuxer.h',
    'MP3FrameParser.h',
]

UNIFIED_SOURCES += [
    'MP3Decoder.cpp',
    'MP3Demuxer.cpp',
    'MP3FrameParser.cpp',
]

FINAL_LIBRARY = 'xul'
```

Fig. 7. A simple moz.build file (dom/media/mp3/moz.build) showing how data can be encoded in this restricted python language. This format is used by Mozilla to describe the build steps that need to be performed.

```
with Files('**'):
    BUG_COMPONENT = ('Toolkit', 'App Update')

DIRS += ['src']

if CONFIG['MOZ_ENABLE_SIGNMAR']:
    DIRS += ['sign', 'verify']
    TEST_DIRS += ['tests']
elif CONFIG['MOZ_VERIFY_MAR_SIGNATURE']:
    DIRS += ['verify']

# If we are building ./sign and ./verify,
# then ./tool must come after it
DIRS += ['tool']
```

Fig. 8. A more advanced moz.build file, with conditionals, subdirectories and bindings between source files and Firefox modules.

Figure 6. We see that the internal state requires an amount of space comparable to the source checkout itself. From quick investigations, we discovered that it is mainly due to dependencies between build steps. further investigations would be required to analyze how this database grows with repository sizes and programming languages, or possibly other significant factors to be discovered.

The ability of Tup to inspect the build graph comes as a nice side-effect of using Tup as a build backend. It shows how using different backends can benefit Mozilla outside of building per se. If Tup ends up not being used by Mozilla, it remains useful for its easily inspect able build graph and also for checking the dependencies declared in the moz.build frontend files.

Besides backends for building, this Mozilla-specific architecture opens the door to other tools instrumenting the build system. For example, Mozilla already uses that build infrastructure to extract a label for each file, in order to automatically relate code changes to Firefox components. It is also used to generate compile-commands.json used by linters and IDE compilation databases [3]. Other tools such as linters and software maintenance tools could be plugged in there, making it low hanging fruit for a practical use case of new research tools.

VI. Firefox as a benchmark for build systems

As was already hinted before, Firefox is our candidate benchmark for comparing build systems. In this section, we explain in further details the structure of Mozilla's build system and the features that catched our attention while looking for a benchmarking project for build systems.

Mozilla's build system is based on simplified python files named moz.build. Examples of these files can be seen in Figure 7 and Figure 8. They list the source files that must be compiled, the name of the libraries or programs that must be produced and various options like compiler flags, headers to be exported and such. The ad-hoc nature of this data format allows Mozilla to encode other kind of information like in Figure 8 where information allows to relate files in the source tree with Firefox components. That example also shows that the build options can depend on configuration values passed in the `CONFIG` dictionary. These values are computed at configuration time, and allow tweaking the build depending on the platform and various other parameters.

The Python format used in moz.build files was picked because it is easy to read and Python is well known among developers, at Mozilla and in general [2]. This input format is parsed by the mozbuild toolchain that generates the right build files according to the selected backend (Makefiles for make, etc.) but it can also perform some complex operations on its own. For example, the backend generates boilerplate unified C++ files, and lists of inputs in separate files to circumvent command-line length limits. The Tup backend performs more operations to work around Tup limitations. For example, it parses some input files to pre-compute the output file names that cannot be simply deduced. We call this step *preprocessing*, and distinguish it from the parsing and generation done by mozbuild. Preprocessing is mostly needed to work around limitations and restrictions in the build backend.

With its custom build description format in Python, Mozilla is independent from a build system in particular. Because Python is a widely known and general purpose programming language, it is possible to adapt it to multiple purposes and in particular to generate instructions for different backends as we did with Tup. As more backends are added, the difficulty to generalize the mozbuild pipeline should lower. Also, in the process of implementing a backend for a build system, features and limitations of said build system appear more clearly and can be collated. With several build systems executing on the same code base, it becomes possible to produce the meaningful comparisons that are not yet available. Our future work will therefore target more build systems and will try to elicit distinctive features for each of them.

Mozilla's custom build framework can be easily adapted to different build systems, and could be used to compare them.

VII. Related Work

Andrey Mokhov et Al. distinguish different kinds of build systems [13]. In Mozilla at the moment, there is a recursive memo/dumb configuration and preprocessing phase. This one is followed by a minimalist build system in the common and strict sense. While not perfect, combining different kinds of build systems provides the ability to work now, and to be perfectible later. This is the approach used by Mozilla in their build system, and makes for a simplified migration, as it allows maintaining two build systems up-to-date side-by-side.

Gligoric et Al. worked on migrating from one build system to another [19]. They extracted build traces from the old one and populated the new build system by factoring out common patterns. Reusing this approach would not take advantage of the existing moz.build architecture used at Mozilla. There is no need to extract traces when the graph is available.

VIII. Reproducibility

This work was performed with the intent to make it fully reproducible. The development environment was managed with the Nix package manager in order to allow a completely reproducible test environment [20]. All the work is available at https://github.com/layus/gecko-tup. The results presented here are all reproducible from the tools provided there.

This also means that our tests were performed within Nix[2], which imposes some strong constraints on the usual Linux builds. Some patches were introduced to circumvent these constraints. This does not invalidate the final results because Firefox was compiled just fine. The issues were mostly with passing custom library paths to a build system that assumes that all the system libraries comply to Filesystem Hierarchy Standard (FHS) [21], which is *not* the case with Nix.

IX. Conclusion

We have conducted an experiment consisting in building Firefox with Tup. It has shown that Tup is a valid replacement for Make on this large real-life software project. While Tup does not generally bring a significant speedup at building, its strength resides in correctness and enforcement of build specifications. We argued that it may be a good investment in the long term, despite the significant refactoring that its introduction requires. Tup is however more efficient than Make at compiling small changesets. It could spare compilation time if such rebuilds are frequent, as could be expected during development on a developer's machine. Beyond its usage as a build system, Tup also provides a straightforward data model that allows in-depth inspection of the build. In the future, we intend to reuse the knowledge and tooling presented here with Tup to compare more build systems, and look for relevant metrics to classify them.

[2]https://nixos.org/nix/

X. Acknowledgments

The authors would like to thank Mike Shal for sharing knowledge about both Tup's and Mozilla's build systems as well as for his careful review of a previous version of this article. Many thanks too to Marie Champenois for producing a better looking Figure 1.

References

[1] Gregory Szorc. *Moving Away from Makefile's*. E-mail. Aug. 22, 2012. URL: https://groups.google.com/forum/#!topic/mozilla.dev.platform/SACOnl-avMs/discussion.

[2] Gregory Szorc. *Moz.Build Files and the Firefox Build System*. Feb. 2013. URL: https://gregoryszorc.com/blog/2013/02/28/moz.build-files-and-the-firefox-build-system/.

[3] The Clang Team. *JSON Compilation Database Format Specification*. URL: https://clang.llvm.org/docs/JSONCompilationDatabase.html (visited on 09/21/2018).

[4] Joshua Cranmer. *Mach Command to Output Clang Compilation Database*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=904572 (visited on 09/21/2018).

[5] Ali Almossawi. *How Maintainable Is the Firefox Codebase?* May 15, 2013. URL: http://almossawi.com/firefox/prose (visited on 10/05/2018).

[6] *The Mozilla Firefox Open Source Project on Open Hub: Languages Page*. URL: https://www.openhub.net/p/firefox/analyses/latest/languages_summary (visited on 10/05/2018).

[7] Stuart I. Feldman. "Make — a Program for Maintaining Computer Programs". In: *Software: Practice and Experience* 9.4 (Apr. 1, 1979), pp. 255–265. ISSN: 1097-024X. DOI: 10.1002/spe.4380090402.

[8] Richard M. Stallman and Roland McGrath. *GNU Make - A Program for Directing Recompilation*. 1991.

[9] Peter Miller. "Recursive Make Considered Harmful". In: *AUUGN*. Vol. 19. AUUGN Journal of AUUG Inc 1. AUUG, Inc., Feb. 1998, pp. 14–25.

[10] Guillaume Maudoux and Kim Mens. "Correct, Efficient, and Tailored: The Future of Build Systems". In: *IEEE Software* 35.2 (2018), pp. 32–37.

[11] Mike Shal. *Tup Build System*. URL: http://gittup.org/tup/index.html (visited on 05/15/2018).

[12] Mike Shal. *Build System Rules and Algorithms*. 2009. URL: http://gittup.org/tup/build_system_rules_and_algorithms.pdf.

[13] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. "Build Systems à La Carte". In: *ICFP* (2018). URL: https://www.microsoft.com/en-us/research/uploads/prod/2018/03/build-systems.pdf.

[14] Ehsan Akhgari. *Unified Builds*. Nov. 2013. URL: https://lists.mozilla.org/pipermail/dev-platform/2013-November/001998.html.

[15] *Add –Build-Plan for 'cargo Build' by Mshal · Pull Request #5301 · Rust-Lang/Cargo*. URL: https://github.com/rust-lang/cargo/pull/5301 (visited on 10/18/2018).

[16] *Support Producing a "Build Plan" without Executing Anything · Issue #3815 · Rust-Lang/Cargo*. URL: https://github.com/rust-lang/cargo/issues/3815 (visited on 10/18/2018).

[17] Mike Shal. *Linking Libxul with Tup and FUSE*. Apr. 2013. URL: http://gittup.org/blog/2013/04/3-linking-libxul-with-tup-and-fuse/.

[18] Bram Adams et al. "Makao". In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference On*. IEEE, 2007, pp. 517–518.

[19] Milos Gligoric et al. "Automated Migration of Build Scripts Using Dynamic Analysis and Search-Based Refactoring". In: *ACM SIGPLAN Notices* 49 (Dec. 2014), pp. 599–616.

[20] Rok Garbas. *Reproducible Development Environments*. Sept. 2015. URL: https://garbas.si/2015/reproducible-development-environments.html.

[21] LSB Workgroup, The Linux Foundation. *Filesystem Hierarchy Standard, Version 3.0*. Mar. 19, 2015. URL: http://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html.