On the Efficiency of Building Large Collections of Software: Modeling, Algorithms, and Experimental Results

Stephen R. Tate[1][0000−0001−9315−2705] and Bo Yuan[1][0000−0002−9862−967X]

Department of Computer Science, UNC Greensboro, Greensboro, NC 27402, USA
srtate@uncg.edu

Abstract. Building software from source code requires a build environment that meets certain requirements, such as the presence of specific compilers, libraries, or other tools. Unfortunately, requirements for different packages can conflict with each other, so it is often impossible to use a single build environment when building a large collection of software. This paper develops techniques to minimize the number of distinct build environments required, and measures the practical impact of our techniques on build time. In particular, we introduce the notion of a "conflict graph," and prove that the problem of minimizing the number of build environments is equivalent to the graph coloring problem on this graph. We explore several heuristic techniques to compute conflict graph colorings, finding solutions that result in surprisingly small sets of build environments. Using Ubuntu 20.04 as our primary experimental dataset, we computed just 4 different environments that were sufficient for building the "Top 500" most popular source packages, and 11 build environments were sufficient for building all 30,646 source packages included in Ubuntu 20.04. Finally, we experimentally evaluate the benefit of these environments by comparing the work required for building the "Top 500" with our environments to the work required using the traditional minimal environment build. We saw that the total work required for building these packages dropped from 139h36m (139 hours and 36 minutes) to 54h18m, a 61% reduction.

Keywords: Build Environments · Large Scale Analysis

1 Introduction

In this paper, we explore the efficiency of building large collections of installable and executable software from source code, focusing on designing environments for performing source code to binary builds. We define the problem from a system independent standpoint, develop models and algorithms at that level, and then we perform experiments and evaluation in the specific setting of Ubuntu version 20.04, which is representative of large collections of system software. We use terminology that is common to Linux distributions, referring to "source packages" that are used to build "binary packages," but even in settings that don't

use packages *per se*, equivalent objects exist even if by other names. To appreciate the scope of the problem in our experimental environment, Ubuntu 20.04 consists of 30,646 source packages that are used to create 62,450 binary packages that end users can install. Using a source package to build binary packages involves certain software requirements, or dependencies, which must be installed to support the build. For example, any source package containing C code will need a C compiler to produce the binaries. Requirements extend beyond the obvious language tools though, and most packages also require certain support libraries be installed to perform the build, and can also require packaging tools and other utilities. Furthermore, dependencies can have their own dependencies, and those dependencies can have dependencies, and so on. The challenges that we address in this paper originally arose not from building binaries, but from performing large scale static analysis on open source software, which has an almost identical set of requirements — for example, running the Clang static analyzer on a source package performs static analysis as a side effect of compilation, so needs all of the build requirements to be installed to perform the analysis. While our motivation was static analysis, we use the "build environment" terminology and problem statement here, since that a more widely-understood setting.

In Debian-based Linux distributions, the build process is typically run using the `pbuilder`[1] tool, which creates a minimal build environment in a chroot jail and adds the necessary dependencies to that environment before starting the build. By adding only the minimal set of dependencies, issues of conflicts are avoided entirely, assuming the package maintainer defined a feasible build environment. While this provides a simple and robust method for performing an isolated build of a single package, when used for building multiple packages the cost of creating each package's minimal build environment becomes very high. Describing work to re-build an entire Debian distribution from sources, Nussbaum reported that some packages required a large amount of time to simply set up the build environment, including a requirement for 485 additional package installations before the build process could even begin for `openoffice.org` [11]. This problem has gotten even worse since Nussbaum's 2009 work, with the version of LibreOffice in Ubuntu 20.04 requiring *904 additional packages*, above and beyond the "build essentials" that all build environments include, and in our tests it took a little over 13 minutes to simply set up before the build could even begin. While `pbuilder` can save created build environments for future use, this is mostly useful for working on a single package that will reuse the exact same build environment on future runs. When the cost of creating package-specific environments became a limiting factor of our work, we began exploring the question of whether we could create generally-useful environments that can be used for large sets of packages. Doing so would greatly improve the efficiency of building or analyzing large sets of packages, and the results of that exploration is the subject of this paper.

When planning to build a large set of packages, it is tempting to think that the right solution is to install all dependencies required by all packages being

[1] https://pbuilder-team.pages.debian.net/pbuilder/

built. Unfortunately, dependencies can conflict with one another, meaning that certain combinations of packages cannot be installed at the same time. For example, in Ubuntu 20.04, building `firefox` requires the package `autoconf2.13` to be installed, while the source package `apache2` lists `autoconf2.13` as a direct conflict. Because of this conflict, it is impossible to set up a single build environment that can support building both `firefox` and `apache2`. While this is a direct and obvious conflict, some conflicts only appear with deeper digging. For example, building `firefox` requires `libcurl4-openssl-dev` and building `git` requires the `libcurl4-gnutls-dev`, and while neither source package lists any conflicts, the two dependencies, `libcurl4-openssl-dev` and `libcurl4-gnutls-dev`, conflict with each other and so induce an indirect conflict in the build environments for `firefox` and `git`. Therefore, to get a complete picture of possible build environments, all dependencies and conflicts, both direct and transitively induced, must be considered.

In this paper, we define a graph that captures the necessary information on dependencies and conflicts for a set of source packages, and show that the graph coloring problem on this graph reflects the corresponds to the design of build environments. Due to the NP-completeness of minimum graph coloring, we cannot efficiently compute optimal solutions, but we explore how well various heuristic approaches perform in practice. Finally, we put these results into practice and test the efficiency of our computed build environments. We make the following specific contributions in this paper:

- Define a "conflict graph" that captures software build dependencies and conflicts, develop efficient algorithms to construct the conflict graph, and explore the properties of this graph for various Ubuntu "Long Term Support" (LTS) releases.
- Show how finding a minimum graph coloring on this graph provides the smallest set of different build environments that need to be created to support building all packages.
- Explore how graph coloring on nested subgraphs supports the ability to analyze subsets of packages (e.g., support analyzing both "the 500 most popular packages" and "the 1000 most popular packages" with a single set of build environments).
- Present experimental results from applying heuristic graph coloring algorithms to these problems using the Ubuntu 20.04 release, and check for evidence of optimality by computing max cliques in these graphs.
- Present experimental results performing software builds, showing that our computed build environments reduce the amount of work required to build the "Top 500" packages in Ubuntu 20.04 by over half.

The problems that we explore are interesting from an abstract modeling and algorithms standpoint, and the reduction and algorithms result in direct practical gains for designing systems for large scale software build and analysis. All code and data reported on in this paper is freely available (see Section 4).

The modeling and coloring results from this paper were originally presented in preliminary form at the ICSOFT 2022 conference [14], and this paper updates

the earlier report by adding checks for optimality by computing max cliques (Section 4.4), adding full experimental tests on the efficiency of using our computed build environments (Section 5), and updating examples throughout to use data from the newer Ubuntu 20.04 release rather than Ubuntu 18.04. The presentation has also been improved throughout and all figures have been updated.

2 Dependency and Conflict Computations

In this section we define the basic terminology and model required for representing packages, dependencies, and conflicts. We simplify some real-world issues for our model, and will discuss those issues and their impact in Section 2.3. *Packages* are sets of files coupled with meta-data, or attributes, that give information about the package. In our model we have a set of *source packages* $\mathcal{S}$ and a set of *binary packages* $\mathcal{B}$, where source packages contain files and information necessary to build binary packages. Attributes for either source or binary packages can include lists of dependencies and conflicts, which for package p we denote by $D(p)$ and $C(p)$, respectively. If $p \in \mathcal{B}$ is a binary package, then $D(p)$ is the set of all binary packages that must be installed any time p is installed in order for p to be functional, and $C(p)$ is the set of all binary packages that cannot be installed at the same time as p. If $p \in \mathcal{S}$ is a source package, then $D(p)$ is the set of binary packages that must be installed and available in the build environment for p, and $C(p)$ is the set of all binary packages that must *not* be installed when building binaries from this source package. In all cases, the $D(p)$ and $C(p)$ definitions specify immediate dependencies and conflicts, and these can induce additional dependencies and conflicts as described in the following subsection.

2.1 Dependency Sets

Package dependencies are defined by package maintainers, and give just immediate dependencies, or what we will call first-level dependencies, which we denote $D_1(p) = D(p)$. Packages in $D_1(p)$ can have their own dependencies, which are called "second-level dependencies," which in turn define "third-level dependencies," and so on. To simplify later cases, we define "level-0 dependencies" of p to be simply the set $\{p\}$, giving the recursive definition

$$D_k(p) = \begin{cases} \{p\} & \text{if } k = 0; \\ \bigcup_{x \in D_{k-1}(p)} D(x) & \text{if } k \geq 1. \end{cases}$$

The full set of dependencies for package p is then

$$D^*(p) = \bigcup_{k \geq 0} D_k(p),$$

and while this union is unbounded in k, only a finite number of levels can add new dependencies since the set of packages is finite. If p is a source package, then all packages in $D^*(p)$ must be installed in order to build binary packages from p.

Since dependencies are directed relations between packages, we can view packages and dependencies as a directed graph (the "dependency graph"). Then $D^*(p)$ is the set of vertices reachable from p in the dependency graph, or equivalently $D^*(p)$ consists of the neighbors of p in the transitive closure of the dependency graph.

While the dependency graph provides a clean abstraction of dependency relations, in our work we work directly with dependency sets. The following algorithm computes a new level of dependencies for package p, where packages in dependencies at prior levels are passed in as parameter E (the "exclusion set"). When called recursively on line 4 below, the new exclusion set E' is at least one element larger than the incoming set E (since E' contains x but E does not), and so the number of levels of recursion is limited by the number of packages (i.e., items that can be added to E). Therefore, recursion must be limited, and the algorithm always completes in a finite number of steps.

```
AllDeps(p, E)
1  S = {p}
2  E' = E ∪ D(p)
3  for x ∈ D(p) − E
4        S = S ∪ AllDeps(x, E')
5  return S
```

Since AllDeps recurses through all levels of dependencies until no additional packages can be added, the end result is $\text{AllDeps}(p, \emptyset) = D^*(p)$ for any package p. The computed dependency set grows monotonically, and once a package is included in the set no additional recursive calls will be made for that package. Therefore, using an efficient set implementation, the AllDeps is very fast. To further improve performance when computing dependency sets of many packages, we cache results for binary packages as they are completed, so we can short-circuit the recursion with pre-computed sets. We discuss this and experimental performance results in Section 4.

2.2 Conflict Sets and Relation

In addition to dependencies, packages can conflict with other packages, which means that they cannot be installed simultaneously. While the Debian package management system has different types of conflicts (e.g., "Conflicts" and "Breaks" attributes in Debian packages), in our model we consider different types of conflicts as the same and refer to them generically as "conflicts." For any package p, we define $C(p)$ to be the set of packages that are listed as conflicting with it, meaning the immediate conflicts. As with $D(p)$, additional indirect conflicts can also be induced through dependencies.

Note that $C(p)$, as defined attributes listed in package p, is not necessarily a symmetric relation between packages. For example, the package maintainer for package p_1 may recognize that there is a conflict with a package p_2, so $p_2 \in C(p_1)$, but the package maintainer for p_2 may not even know that package

p_1 exists, so would not list p_1 as a conflict and thus $p_1 \notin C(p_2)$. Regardless of whether or not both packages recognize the conflict, if it exists in either direction then the packages cannot be installed simultaneously. We take care of both the possible lack of symmetry and indirect conflicts from dependencies in the following definition.

$$C^*(p) = \{r \mid r \in C(d) \text{ for some } d \in D^*(p) \quad \text{or} \\ d \in C(r) \text{ for some } d \in D^*(p)\}. \tag{1}$$

Note that $C^*(p)$ is symmetric, meaning that $r \in C^*(p)$ if and only if $p \in C^*(r)$. The package p in this definition can be either a source package or a binary package, and if p_1 and p_2 are source packages with $p_1 \in C^*(p_2)$ that means that their build environments are incompatible (some package required to build p_1 conflicts with some package required to build p_2). Conversely, if $p_1 \notin C^*(p_2)$ then the build environments are compatible: all packages in $D^*(p_1) \cup D^*(p_2)$ can be installed together, and that environment will support building binary packages from both p_1 and p_2.

2.3 Model vs Real World

Our model captures the basic ideas of dependencies and conflicts, while avoiding some complications found in real-world package management systems. In this section we summarize the main differences between our model and the Debian package management system that inspires this work.

Disjunctions in dependencies: While our model uses a set of packages $D(p)$ to represent a conjunction of dependencies, the Debian package manager allows each dependency to be satisfied in multiple ways – a disjunction of packages, which we call an "or-list" for that dependency. For example, in Ubuntu 20.04, the `weechat` package has a single dependency, which is satisfied by either of two packages: `weechat-curses` or `weechat-headless`. We keep and propagate these disjunctions up to the level of the source package when computing $C^*(p)$, and then select a set of non-conflicting packages to satisfy each disjunction in left-to-right preference order. This is the same prioritization used by the official Debian build systems, as described in the Debian Policy Manual: "To avoid inconsistency between repeated builds of a package, the autobuilders will default to selecting the first alternative, after reducing any architecture-specific restrictions for the build architecture in question" [8]. Our code first removes all disjunctions that are met by some other (possibly transitively-induced) dependency, and then performs an exhaustive backtracking search over disjunctions to satisfy dependencies, which can take exponential time in the worst case. Other authors have shown that the basic co-installability question for packages is NP-complete due to these disjunctions (see the "Related Work" section), but in our experiments we found that real-world package data resulted in quick dependency resolution in practice, with backtracking in our search being very rare.

"Provides" pseudo-packages: Similar to explicitly providing alternatives for a dependency, Debian allows for certain package names to represent "virtual

packages" which can be satisfied by a number of real packages. For example, in Ubuntu 20.04, `lpr` is both a binary package and a virtual package, and the virtual package is provided by not only the binary package named `lpr` but also by packages `lprng` and `cups-bsd` which are drop-in replacements for the `lpr` package. Our tools treat virtual packages as if they were disjunctions, described above, prioritizing a package with the given name over other packages which provide equivalent functionality.

Versions requirements in dependencies: Dependencies can include version numbers as well as package names. For example, in Ubuntu 20.04 the `libfwsi1` requires `libc6` version 2.14 or greater. While these can technically specify arbitrary version requirements, in Ubuntu 20.04 all version requirements are met with the current ("candidate") version. For example, the base `libc6` package distributed in Ubuntu 20.04 is version 2.31, so the stated version requirement is clearly met. Version requirements seem to be mostly relevant for users who attempt to install newer versions of packages (with newer versions specified in dependencies) on older base systems. Since we did not find any relevant version-specific dependencies when sticking with just base distribution packages, we ignore version requirements in our study.

Recommended packages: Dependencies and conflicts aren't the only relations between packages, and packages can also "Recommend" or "Suggest" other packages. Since these are not necessary in a build environment, our tools ignore these packages.

3 The Conflict Graph and Coloring

In this section we define the "conflict graph" and show that there is a one-to-one correspondence between valid vertex colorings of this graph and feasible sets of build environments. This provides a standard and well-understood graph theory context for understanding sets of build environments.

The conflict graph is an undirected graph that has one vertex for each source package, and each edge represents a conflict in the minimum build environments for edge's two endpoints (source packages). In particular, we define the graph $G = (V, E)$ where the vertex set $V = \mathcal{S}$ (the set of source packages), and

$$E = \{(p_1, p_2) \,|\, p_1, p_2 \in \mathcal{S} \text{ and } p_1 \in C^*(p_2)\}.$$

Since vertices are source packages, we interchangeably use the terms "vertex" and "source package" in the rest of this paper. If two vertices are connected in this graph, it means that there are incompatibilities in the build environments for the two source packages, so there is no build environment that can be used for both.

Figure 1 illustrates a conflict graph for four source packages in Ubuntu 20.04. The graph comprises nodes and edges, and additional information about specific dependencies and conflicts is provided in the labels. Specifically, the dependencies for each package p in $D^*(p)$ are displayed, along with conflicts among packages listed as dependencies, indicating which packages are incompatible with each other.

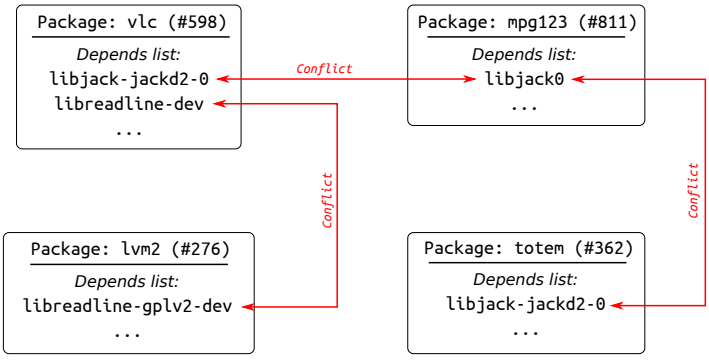

Fig. 1. Example of a conflict graph from Ubuntu 20.04 source packages.

3.1 Colorings and Build Environments

A k-coloring of a graph $G = (V, E)$ is a mapping from vertices to a set of k colors, $c : V \to \{1, \ldots, k\}$, such that every edge in G has endpoints that have been assigned different colors. In other words, for every $(u, v) \in E$, we have $c(u) \neq c(v)$. The usual goal in finding a graph coloring is to minimize the number of colors k required, and the minimum k for a graph G is called the *chromatic number* of the graph, denoted $\chi(G) = k$.

Colorings of the conflict graph have a direct correspondence with sets of build environments. Specifically, consider a k-coloring on a conflict graph: two source packages that have incompatible build environments due to a conflict correspond to two vertices that are connected by an edge, so in any valid coloring those source packages will be assigned different colors. We will associate each color with a distinct build environment, so this property ensures that two source packages with incompatible build environments in fact use different build environments. We now prove that colorings on the conflict graph have a one-to-one correspondence with sets of build environments for the source packages.

Lemma 1. *Every set of k distinct build environments that can be used to build all source packages can be used to define a k-coloring on the conflict graph.*

Proof. For every $i = 1, \cdots, k$, let P_i denote the set of binary packages included in the ith build environment, and define a coloring c that assigns color i to any vertex (source package) that uses this build environment. Since all source packages have a build environment, every vertex is assigned a color. To see that this is a valid coloring of the conflict graph, consider two vertices v_1 and v_2 that are connected by an edge (v_1, v_2) in the conflict graph, meaning that $v_1 \in C^*(v_2)$. By (1) it follows that there is a $d_1 \in D^*(v_1)$ and $d_2 \in D^*(v_2)$ such that either $d_1 \in C(d_2)$ or $d_2 \in C(d_1)$. Therefore, if v_1 uses build environment P_a and v_2 uses build environment P_b, then $d_1 \in P_a$ and $d_1 \notin P_b$ and so $P_a \neq P_b$. Since v_1 and v_2 use different build environments, they must have different colors in c.

As this holds for any edge (v_1, v_2) in the conflict graph, and there are k build environments, c is a valid k-coloring of the conflict graph. ■

Lemma 2. *Every k-coloring of the conflict graph can be used to create a set of k distinct build environments that is sufficient to build all source packages.*

Proof. Let $c : V \to \{1, \ldots, k\}$ be a k-coloring of conflict graph $G = (V, E)$. The k-coloring partitions the vertex set V, and we can define $V_i = \{v \mid c(v) = i\}$ (this is often called a "color class"). Next, for each $i = 1, \ldots, k$, define a set of binary packages $P_i = \cup_{v \in V_i} D^*(v)$. We claim that for every source package $v \in V_i$, P_i is a valid and feasible build environment for that source package. The fact that P_i is sufficient follows directly from the definition, since that requires all dependencies of any $v \in V_i$ to be included in P_i.

For feasibility, we need to show that all packages in P_i can be installed simultaneously with no conflicts. For the sake of contradiction, assume that there are conflicting packages $p_1, p_2 \in P_i$ with $p_1 \in C(p_2)$. The inclusion of p_1 and p_2 in P_i must be the result of two source packages $v_1, v_2 \in V_i$ with $p_1 \in D^*(v_1)$ and $p_2 \in D^*(v_2)$. Since $p_1 \in C(p_2)$, it follows that $v_1 \in C^*(v_2)$ by (1), and so there is an edge (v_1, v_2) in the conflict graph. However, since v_1 and v_2 are in the same V_i partition component, they must both have color i which violates the basic coloring requirement. This contradiction completes the proof. ■

The following theorem follows directly from the two preceding lemmas.

Theorem 1. *The conflict graph has a k-coloring if and only if there is a set of k distinct build environments that is sufficient to build all source packages.*

The proofs above are constructive arguments that provide efficient algorithms for converting from a k-coloring of the conflict graph to a set of build environments, but is this the best way to solve the problem? k-coloring is NP-hard, and does not have efficient approximation algorithms (unless P=NP), so is this a useful reduction? In other words, the question remains of whether finding build environments might in fact be easier than graph coloring – is there some sort of structure to conflict graphs that would lead to efficient solutions, even though the minimum graph coloring problem on general graphs is NP-complete?

Unfortunately, the answer to this question is "no." For an arbitrary graph G we can easily create a set of source packages and conflicts for which the conflict graph is G simply by making a distinct source-to-source conflict for each edge in G. Note that we don't even need to consider binary packages and dependencies for this construction. To be precise about this, using the notation from Section 2 (where $\mathcal{S}$ is a set of source packages, $\mathcal{B}$ is a set of binary packages, D is a dependency function, and C is a conflict function), we define a decision problem (language) MIN-BUILDENV= $\{\langle \mathcal{S}, \mathcal{B}, D, C, k\rangle \mid$ there exist a set of k feasible build environments that is sufficient for building all source packages in $\mathcal{S}\}$. The construction which was described at the beginning of this paragraph is a reduction from MIN-COLOR= $\{\langle G, k\rangle \mid$ there is a valid k-coloring of $G\}$ to MIN-BUILDENV. Since MIN-COLOR is a known NP-complete problem (problem

[GT4] in Garey and Johnson [6]). This leads to the following theorem, which we state without further proof.

Theorem 2. *MIN-BUILDENV is NP-complete.*

This result is discouraging as far as worst-case complexity of MIN-BUILDENV, but instances created in the reduction are somewhat unnatural and real-world instances may very well be easy cases that are in fact tractable. We explore properties of conflict graphs derived from real-world software in Section 4, but leave open the question of whether typical real-world instances can be solved efficiently. Before getting to the experimental results, we define and discuss an interesting variant of our problem.

3.2 Nested Sets of Source Packages

As mentioned in Section 1, our work in creating a formal framework in which to study this problem arose from a project performing large-scale analysis of open source software to search for security vulnerabilities. To maximize the practical impact of our work, we concentrate primarily on the most widely used packages, which we gauge from the now defunct "Ubuntu Popularity Contest" project [15]. We develop our techniques using a small set of packages, and then test on the most popular 100 Ubuntu packages. If that shows promising results, we might devote more computational resources and analyze the most popular 500, 1000, or even 5000 packages. In this process, we work with increasingly larger nested sets of source packages, which motivates an extended version of our build environment definition problem. For example, if we set up a minimal set of build environments for the most popular 500 packages, can we use those environments (with possible extensions) for the most popular 1000 packages? Unfortunately, it is impossible to optimize for both 500 and the expanded set of 1000 simultaneously.

To understand the problem, we will revisit the example in Figure 1. The numbers beside each package name refer to the position of the package in the Ubuntu Popularity Contest ranking, so `lvm2` is the 276th most popular package, `vlc` is the 598th most popular package, and so on. First consider what would happen if we created build environments for the most popular 500 packages, which would include both `lvm2` and `totem` in our example. Since there are no conflicts between these two packages, the subgraph consisting of just those two packages can be colored with a single color, meaning that a single build environment can be constructed that can be used to build both `lvm2` and `totem`. When we expand this to the "Top 1000 packages," we end up with the full 4-vertex conflict graph shown in Figure 1. To reuse the build environments we previously constructed, we would need to extend the existing coloring (where `lvm2` and `totem` are given the same color) into a coloring for the entire 4-vertex graph. Unfortunately, when we retain those colors we require 3 colors (or 3 different build environments) for the 4-vertex graph. On the other hand, if we were to color the 4-vertex graph from scratch we could do so with just 2 colors. In other

words, by trying to keep the same build environments from the "Top 500 packages" solution, we are forced to take a sub-optimal solution to the "Top 1000 packages" case.

Since extending from the smaller set of packages to the larger doesn't work, can we go in the other direction? Can we compute a solution to the larger problem and then restrict that solution to the smaller set? Again, referring to Figure 1, we can see that any 2-coloring of this graph results in the two more popular packages, `lvm2` and `totem`, being assigned different colors. This means that when we restrict our larger solution to just the two most popular packages, we are forced to use two distinct build environments when a single build environment would suffice.

Using a larger set of build environments not only increases storage requirements, but also negatively impacts time required for running a large set of builds due to caching. If we build a package using build environment A, and can reuse that same build environment for a second package, many of the files in build environment A will be cached in memory already, leading to a faster build for the second package. If the second package used a separate build environment B, as in the example in the previous paragraph, then the files in environment B would need to be loaded from disk in building the second package, giving a performance hit.

In our work, we have prioritized creating the smallest set of build environments for each of the nested sets of source packages, and do not try to reuse environments from one collection of source packages to the next. Our experiments show that the amount of extra space required to host multiple independently-computed sets of build environments is modest, and we feel that the gains while working within that collection outweigh the costs. We leave further optimization in this setting to future work.

3.3 Conflict Graph Simplification

When a conflict graph is created and examined, it quickly becomes clear that there are some simplifications that can be made to reduce the size of the graph while still maintaining the correspondence between coloring and build environments. The most obvious is that approximately two-thirds of all source packages in modern Ubuntu releases have no conflicts at all, either direct or induced indirectly, so can be built in any build environment simply by including the necessary conflict-free dependencies. In terms of the conflict graph, these source packages exist as isolated vertices in the conflict graph. Since these vertices do not affect the coloring, they can be removed from the graph and then assigned arbitrary colors at the end.

More generally, we can merge isomorphic vertices into a single vertex. If source packages p_1 and p_2 have the same set of conflicting source packages, meaning that $C^*(p_1) \cap \mathcal{S} = C^*(p_2) \cap \mathcal{S}$, then vertices p_1 and p_2 have the same neighbors and any color that is valid for p_1 will also work for p_2 without any other changes necessary in the graph coloring. Because of this, we merge isomorphic vertices, repeating this process until a fixed point is reached. As we'll see in the

next section, this reduces the size of the graph we need to color by over 90%, which greatly improves both the speed and effectiveness of the heuristic graph coloring algorithms that we use.

This process is illustrated in Figure 2, where we show two steps of merging isomorphic vertices. In this example, source packages A and B each conflict with all of C, D, and E, giving the conflict graph drawn on the left. In the first step, we note that vertices A and B have exactly the same set of neighbors (i.e., conflicting source packages), so are merged into a single vertex that represents both A and B. In the second step, we see that C, D, and E are similarly isomorphic, so are merged into a single vertex. It is clear that this final graph has an optimal 2-coloring, which can be translated back to the original, larger graph.

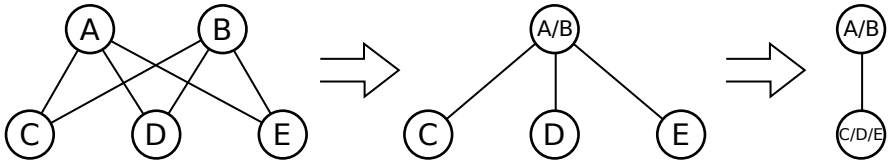

Fig. 2. Illustration of Graph Simplification

4 Experimental Results

In this section, we present experimental results that we obtained by analyzing four recent Ubuntu LTS releases. We created virtual machines for each of the releases in that time frame, with the main, restricted, universe, and multiverse components loaded into the package database, and with updates turned off so that only versions in the original distribution were included. We next developed software to extract dependency and conflict information from package information using Python and the Python APT Library[2]. This worked well for Ubuntu releases 16.04 and later, but the version of `python-apt` included with 14.04 lacked key features that we relied on. To analyze the Ubuntu 14.04 release, we created a docker container based on Ubuntu 20.04 with all of the Python packages needed to create the conflict graph, including a sufficiently modern version of `python-apt`. We then manually removed all of the Ubuntu 20.04 package information files from that Docker container, and replaced them with files extracted from a Ubuntu 14.04 installation, and in that way successfully constructed the conflict graph for Ubuntu 14.04. Our code and results from this set of Ubuntu distributions is available in a GitHub repository under an open source license[3].

[2] https://apt-team.pages.debian.net/python-apt/library
[3] https://github.com/srtate/BuildEnvAnalysis

First, we examine basic properties of dependencies and conflict graphs, as well as effect graph simplification as described in Section 3.3, to gain insight into the size and structure of real-world data. Then in the following sections, we report on results from running heuristic graph coloring algorithms on the generated graphs, and discuss what that means for setting up build environments.

4.1 Graphs from Ubuntu LTS Releases

Ubuntu version:	14.04	16.04	18.04	20.04
Buildable SPKGS	22,028	25,401	28,886	30,646
Full graph edges	207,894	214,982	376,028	387,175
Full graph density	0.0009	0.0007	0.0009	0.0008
Simplified graph vertices	1,646	1,943	1,476	1,770
Simplified graph edges	37,087	45,992	43,363	51,492
Simplified graph density	0.027	0.024	0.040	0.033

Table 1. Basic graph metrics for Ubuntu releases (table appeared in [14])

We first consider the basic properties of the conflict graphs constructed from four major long-term-support (LTS) releases of the Ubuntu Linux distribution, which were released at two year intervals from 2014 to 2020. To gain insight into what to expect in future releases, we evaluate some fundamental graph metrics on these releases, examining what has remained consistent and what has changed over time. Table 1 presents the results for both the complete conflict graphs and the simplified graphs, providing information on their sizes and density measures. The column labeled "Buildable SPKGS" shows the number of buildable source packages with each release. It is worth noting that both the 16.04 and 18.04 release had six source packages that were not buildable due to conflicts in their dependency attributes. These issues were resolved through updates to the LTS release, but for the sake of consistency, we excluded these unbuildable source packages from our analysis of non-updated releases.

As can be seen in the table, the number of source packages has increased with every new release, giving an overall 39% increase from 14.04 to 20.04. Our graph simplification algorithm, as described in section 3.3, consistently reduces the number of vertices in the conflict graph by between 92% and 95%. This significant size reduction allows our heuristic graph coloring algorithms to run significantly faster and with higher success rates, as we will see in Section 4.2.

Also of interest is the structure and complexity of the dependencies and conflicts. We originally predicted that dependency chains would be relatively short, and while the vast majority of dependency chains are under 10 links long, in our work on the 20.04 release we found one dependency chain of length 18, for package `node-brfs`. We found, unsurprisingly, a large number of packages

with mutual dependencies, although some came from the same source package, so it's unclear why separate binary packages are built if they must always be installed together (e.g., `tasksel` and `tasksel-data` depend on each other, and are both built from source package `tasksel`). While such mutual dependencies give cycles of length two in the dependency graph, there are simple cycles of varying lengths larger than two as well — for example, `console-setup` depends on `console-setup-linux`, which depends on `kbd`, which depends on `console-setup`.

With this understanding of release sizes and metrics, we next report our experimental results using heuristic graph coloring algorithms on the constructed conflict graphs.

4.2 Coloring Results for Ubuntu 20.04

Since graph coloring is an NP-hard problem, the graphs constructed from Ubuntu distributions, even after simplification, are far too large for any exact optimal graph coloring algorithm to be practical. Therefore, we need to rely on approximate (heuristic) graph coloring algorithms, and we utilize the suite of graph coloring algorithms created by Joseph Culberson[4]. This software supports a variety of heuristics, ranging from a simple greedy algorithm to versions that use heuristics and randomization to find better colorings. Random seeds to be provided on the command line, which enables us to script a large number of many runs with varying random seeds. This software was designed for the DIMACS challenges on graph coloring, readings graphs in the "DIMACS standard format," so our package analysis software outputs the conflict graphs in this format. Additionally, we output "translation table" that allows us to convert between a generic vertex number and its corresponding source package name.

We first considered two versions of graphs that represent all source packages: the full conflict graph and the simplified version computed as described in Section 3.3. Note that for the simplified graphs, the translation table mapping package names to vertices is many-to-one, so vertices are not synonymous with individual source packages. We automated the process of running the coloring algorithms with different random seeds and different heuristic options, and allowed the coloring programs to run for up to a full 24-hour day on a Linux system with an Intel i7-7700 processor. For both the full and simplified graphs generated from the Ubuntu 20.04 distribution, the coloring software found colorings using as few as 11 colors, meaning that 11 distinct build environments are sufficient to build all 30,646 source packages. Since these are approximation algorithms, there's no way to tell from this software if 11 is the minimum possible number of build environments, but subsequent tests reported in Section 4.4 show that this is in fact an optimal solution.

Comparing the performance and success of coloring the full graph versus the simplified graph shows the value of graph simplification: the colorings found on the simplified graph translate directly to the full graph, but the reduced size

[4] http://webdocs.cs.ualberta.ca/~joe/Coloring/

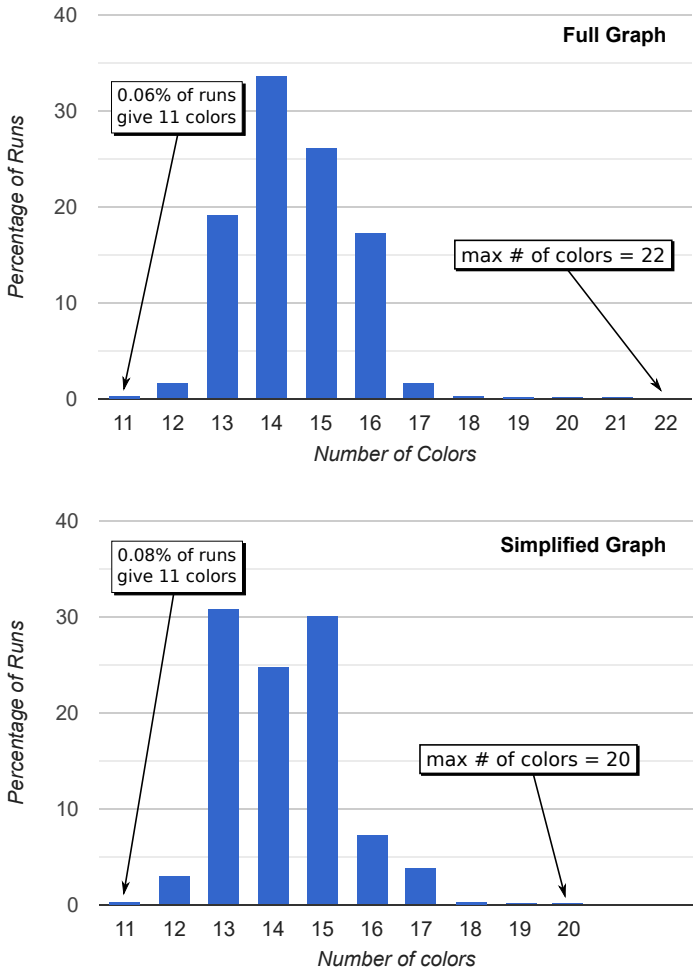

Fig. 3. Number of colors found by heuristic run, as a percentage of all runs.

allowed the coloring software to run much faster and explore more options with more random seeds. We completed over a million (specifically, 1,050,000) runs on the simplified graph in 24 hours, while we could only complete 23,835 runs on the full graph. Working with a smaller graph also increased the probability that the the heuristic algorithm would find small colorings, avoiding getting stuck in parts of the graph that lead to using larger numbers of colors. Figure 3 shows histograms of the colors found over all runs, for both the simplified and the full graphs. Notice that with the simplified graph the values are skewed more to the left, meaning colorings with fewer colors. The range for the simplified graph is also lower, with the coloring software producing colorings ranging from 11 to 20 colors on the simplified graph, and 11 to 22 colors on the full graph.

While the percentage of runs finding the smallest (11 color) solution is only slightly higher for the simplified graph (0.0806% of runs) than the full graph (0.0587% of runs), that small advantage coupled with the much higher rate of testing graphs, means that the simplified graph found the smallest coloring much faster than using the full graph. More specifically, the first coloring using 11 colors was found in just under 5 minutes on the simplified graph, but an 11-color result on the full graph was not obtained for 99 minutes.

4.3 Subgraph Colorings

	Top 500	Top 1000	Top 2000	Top 4000	All SPKGS
Vertices	117	198	355	594	1,770
Edges	151	375	2,388	6,385	51,492
Density	0.022	0.019	0.038	0.036	0.033
Best coloring	4	4	6	7	11

Table 2. Basic metrics for Ubuntu 20.04 top-X subgraphs (table appeared in [14])

Next, we construct colorings for nested subgraphs, as described in Section 3.2. To construct these graphs for the Ubuntu 20.04 distribution, we first used the "Ubuntu Popularity Contest" project data to find the top 500, 1000, 2000, and 4000 source packages. Note that this is not as simple as just taking the first names from the popularity contest ranking for two reasons: First, not all packages listed are standard packages in the Ubuntu release we are interested in (20.04 in this case), and second, the ranking is for binary packages, not source packages. To find our lists, we first filtered the popularity contest list to include only binary packages that are a part of the Ubuntu 20.04 release, then we mapped binary package names to the source package used to build that package, and finally we removed all but the first occurrence of each source package (since a source package can build multiple binary packages, and it is common for multiple binary packages for the same source package to be in the "Top X" lists). As a result of

this pre-processing, we obtained a ranked list of source packages used in Ubuntu 20.04 from which we could extract the "Top X" lists.

Next, given the Top 500 source packages, we identified the vertices in the simplified conflict graph corresponding to those packages, removed duplicates, and then computed the subgraph induced by those vertices. We repeated this for the top 1000, 2000, and 4000 source packages. Given these graphs, we ran 105,000 iterations of the graph coloring algorithm on each to determine the smallest coloring we could find in that amount of time. The results, showing graph sizes and the best coloring we found, are in Table 2.

When we started this project, we expected that the main, most popular Ubuntu packages would share a somewhat similar mainstream build environment with few conflicts, leaving most conflicts to arise from more esoteric packages that used unusual libraries. However, we were surprised to find that even in the smallest (Top 500) list, there were over a hundred non-isomorphic conflict sets (final vertices after simplification), and a fairly consistent graph density of 0.02-0.04 in each graph. The colorings we found do reflect increasing levels of conflicts when more, less popular packages are included, growing from just 4 colors (build environments) used for the Top 500 and Top 1000 lists, up to the 11 colors required for all packages.

As a final note, this project was motivated by our experience in an earlier project in which we were performing large scale static analysis on the Top 1000 source packages in Ubuntu 18.04. When we ran into build environment conflicts, we handled this problem in an *ad hoc* way, resulting in around a dozen build environments for the Top 1000 packages. Developing a formal foundation for creating these build environments, as we report in this paper, reduces the number of build environments for that set of packages to just 4 distinct environments. This is a huge improvement in terms of managing the 1000 runs of the static analyzer, and as we will see in Section 5 it also resulted in a large efficiency when performing the actual builds.

4.4 Lower Bounds Using Max Clique

The graph colorings reported in the previous section were computed using heuristic algorithms, so in this section we consider the question "are the colorings found above the best possible?" Even when the chromatic number of a graph is small, determining the optimal coloring can be computationally intractable. Consider that every planar graph is 4-colorable, but determining if a planar graph is 3-colorable is NP-complete (determining if it is 2-colorable is easy) [6].

The maximum clique problem is the problem of finding the largest completely connected subgraph in an input graph. Since every vertex in a completely connected subgraph is connected to every other vertex in that subgraph, when coloring such a graph each vertex in the clique must have a different color, and so the size of a maximum clique is a lower bound for the number of colors required to color that graph. Furthermore, for inputs where the size of the maximum clique is guaranteed to be bounded by a constant k, we can find the largest clique in time $O(n^k)$. In our case, since we found a valid coloring of size 4 in the "Top 500"

graph, we know that the maximum clique can be no larger than 4 vertices and so we could find a maximum clique in that graph in $O(n^4)$ time. Unfortunately, for the full graph the brute-force search for cliques of size 11 would take time $\Theta(n^{11})$, so we need a faster algorithm.

The graphs we are dealing with are quite sparse (see Table 1), and maximum clique algorithms designed for sparse graphs proved to be very successful. In particular, Pattabiraman *et al* designed a max clique algorithm for sparse graphs, specifically looking at applications with Web connectivity graphs [12]. Importantly, their algorithms provide optimal solutions, not heuristic approximations, and gain speed improvements through novel pruning techniques during the search for a clique. The authors produced software implementing their algorithm and made it publicly available, and we used this on the conflict graphs generated as part of this project.[5] The results are shown in Table 3.

	Top 500	Top 1000	Top 2000	Top 4000	All SPKGS
Best coloring	4	4	6	7	11
Max clique	3	4	6	6	11

Table 3. Computing coloring lower bounds using max clique

As can be seen from the results, the size of the max clique matched the number of colors in our graph coloring for three of the graphs: the Top 1000, Top 2000, and full graphs. That means that our graph colorings are optimal in those cases, and due to the one-to-one correspondence between coloring the conflict graph and designing build environments, we can conclude, for example, that there is no set of fewer than 6 build environments that is sufficient for building all Top 2000 source packages (with the caveat that there may be better solutions with different handling of dependency disjunctions – see the open problems). For two graphs, the "Top 500" and "Top 4000" graphs, the size of the largest clique was one smaller than the number of colors in the best coloring we found. This means that there is still some uncertainty about whether 3 or 4 build environments is sufficient for the "Top 500" packages, but even in the two cases in which we can't assert optimality, we are within one of an optimal solution.

5 Build Times in Practice

The previous sections have developed a model and associated algorithms for designing build environments. In this section, we report on experiments performed to measure the practical impact of these techniques.

[5] Software available at http://cucis.ece.northwestern.edu/projects/MAXCLIQUE/

5.1 Build Experiment Setting

To test the efficiency of our computed build environments, we used a server with dual Intel Xeon E5-2623 v3 CPUs, each with 4 true cores with hyperthreading, bringing the total number of threads to 8 per CPU or 16 total. The system has 64 GB of RAM with a small 64 GB SSD-based filesystem for the main filesystem (storing all executables) and a larger 2.8 TB HDD-based filesystem used for holding source code and performing the builds.

We performed all experiments using the "Top 500" source packages from Ubuntu 20.04, but with a few packages removed (see "Practical Challenges" below). We compared the standard build process, using `pbuilder`, with the use of four pre-built environments as computed by the coloring-based algorithm of this paper (described in Section 4.3). While source packages with no conflicts could be assigned any color (build environment) we put them all into "build group 1" resulting in a very large build environment. The four environments we constructed had the following sizes:

	SPKGs using	BPKGs installed	Size (MB)
Environment 1	355	3,976	11,905
Environment 2	21	1,928	5,326
Environment 3	27	1,816	5,422
Environment 4	97	1,140	2,716

The `pbuilder` tests used a pre-constructed base environment, which included all essential and build-essential packages, but then followed the standard Debian process after that: a chroot environment is created for the base environment on top of which the dependencies are installed, then source packages are unpacked and binary packages are built after dropping privileges for the build. Building with our computed build environments is similar, except the four different build environments are pre-loaded with the union of all dependencies for the supported source packages, and so no additional build dependencies have to be installed for each build – we simply unpack the source packages and then perform the build in a chroot environment after dropping privileges.

For both techniques, we used GNU `parallel` to perform 12 builds in parallel, but restricted each individual build to a single-threaded make (i.e., "`-j 1`" provided to `make`). The decision to do single-threaded `make` was so that each package build used a similar set of resources, avoiding the inconsistency that results from some packages being able to make use of concurrent builds and others not. The choice of 12 parallel builds was somewhat arbitrary, but was determined to be a good trade-off between using available threads for building and leaving some excess capacity for slack and system management functions.

5.2 Practical Challenges

As we started performing the actual builds, it quickly became clear that there were packaging issues that went beyond the modeling simplifications that described in Section 2.3. Here are some of the issues we encountered:

Bad package definitions: Some source packages had errors in dependencies and conflicts, resulting in unbuildable packages under either `pbuilder` or our build environments. For example, the `gconf-editor` source package depends on binary package `gnome-doc-utils`, but `gnome-doc-utils` is not packaged for Ubuntu 20.04. This makes the `gconf-editor` package impossible to build in a pure Ubuntu 20.04 environment. Given this fact, it is unclear how the Ubuntu maintainers created the binary package for `gconf-editor`, but the binary could have been a pre-20.04 package included with 20.04, or the maintainers could have used additional package sources to provide `gnome-doc-utils`. Regardless, since it is impossible to build with a pure 20.04 system, this is a packaging bug. Earlier work by Nussbaum [11] explored using a cluster to test all source package definitions for exactly this kind of bug, but this is apparently not a part of the Ubuntu release quality assurance process.

Builds that require additional system resources: Some package builds hung when using either `pbuilder` or our build environments due to use of resources beyond files. For example, `gnome-keyring` could not be built in a batch process with standard command line arguments, since some of the tests required terminal interaction. Other packages failed to access network resources, including attempts to listen on network ports that were already in use on the server when testing. Almost all of these kinds of problems failed in running tests after the binaries were built, and so one potential solution is to exclude testing from the build process. This could potentially speed up the process too, since some packages spent a significant amount of computational time performing tests.

Problems with build environments that are too large: Our original assumption was that build environments couldn't be "too large" as long as no conflicting packages were installed. However, this turned out to be a false assumption. A specific problem we encountered came from a configuration script used to set up the build, which enabled some tests based on the availability of testing tools. Since our build environment is larger than necessary, our build attempted to run tests that were not run in the minimal environment used by `pbuilder`. A minor consequence of the extra tests being run is that the build can take longer than under the `pbuilder` minimal environment. However, a much more serious problem is that some packages *failed* these additional tests, so did not produce the desired binary package. It's debatable what the actual error is in this case. Is it the failed tests, meaning the built executables have bugs that were not caught in the `pbuilder` build? Or is the problem that the testing tools should have been omitted from the build environment, as they were in the minimal `pbuilder` environment? If the latter is the case, then since these tools cause the build to fail, they should be included in the source package "Conflicts" list to ensure that they are not installed when the package is being built.

The three issues described above caused real problems, but were relatively rare, causing problems with less than 6% of source packages. Since the goal of this project is to test the efficiency of build environments, we did not manually tweak the builds to correct these individual issues, and we just omitted those packages from our test set. In the end, we were left with 472 packages in the

Top 500 that built cleanly and without any manual intervention, and the results reported below are for those 472 packages.

5.3 Results

After adjusting for the issues described in the previous section, for the 472 packages in our test set we saw a substantial improvement by using our build environments, both from the standpoint of total work (sum of individual package build times) and an overall elapsed wall-clock time. When looking at the total work performed over our 12 parallel jobs, `pbuilder` required 139h36s (139 hours and 36 minutes), while using our build environments required only 54h18s, a 61% improvement and saving over 85 hours of CPU time. The overall elapsed time also showed improvement, but less dramatically, taking 25h05m with `pbuilder` and 19h55s with our build environments. The reason that the overall elapsed time did not show as dramatic an improvement as the total work was that a few packages took a *very* long time to build, with the longest being `libreoffice` which took over 19 hours to build by either technique. This build process long outlasted other builds, with the final 8 hours of the test builds consisting of just that one build running. Since the overall elapsed time was dominated by this one build, which only improved slightly with the pre-constructed build environments, the overall time had a more modest improvement than total work. Since our goal was to evaluate change in total work based on using the build environments constructed from our coloring algorithm, we leave exploration of scheduling improvements to future work. Some improvement could certainly be obtained by scheduling jobs in order of decreasing work (so `libreoffice` would start first), but that would require *a priori* knowledge of build times. Alternatively, we could revisit our decision to limit each package to a single-threaded make, and work on balancing parallel builds with parallelization over packages.

Improvement in mean and median times: For the packages in our test set, the mean package build time using `pbuilder` was 17m44s (17 minutes and 44 seconds), while the median was 8m23s. Using our coloring-determined environments reduced the mean build time to 6m54s (a 61% decrease) and the median to just 45 seconds (a 91% improvement). The difference between the mean and median, as well as the dramatic improvement in the median time, emphasizes the fact that while there are some huge packages that raise the average, most packages are quite small, taking less than 45 seconds to build once the environment is set up. For these small packages the vast majority of the time `pbuilder` requires is used to simply create the individualized build environments.

Package with largest time improvement: In terms of improvement in absolute time, the greatest improvement was for the `gcc-10` package. This package required a little over 9 hours to build using `pbuilder`, but only 7 hours and 51 minutes to build in our environments. The 70 minute decrease in time was substantially more than just the time savings from creating the build environment, since that only took approximately 21 minutes in the `pbuilder` trial. The reason for the additional 50 minutes in time savings is not clear at this time. We do know that both `pbuilder` and our build environments produced the same

binary packages, which passed all tests, so everything was correctly built. The `gcc-10` build is extremely complex, with a build log that is over 200,000 lines of text, and we are currently combing through that log for additional clues.

Package with the largest percentage improvement: Some packages had very dramatic speed-ups. For example, the `language-pack-gnome-en` source package doesn't really require a "build" at all — it simply packages a set of text files with the directory paths needed for the binary install, so the "build" really just consists of copying files and creating the package archives. This "build" in a pre-constructed environment takes around 2 seconds. However, `pbuilder` still needs to create the build environment, including the packaging tools (like `debhelper`), resulting in a total time of 316 seconds. Using the pre-constructed environment resulted in a 99.4% reduction in time for this package.

Slower builds: Our original focus on dependencies led to a mistaken belief that as long as all the dependencies were present, the runtime should be mostly independent of whether other, non-required packages were installed. However, it turned out that the build process for some packages slowed down in larger environments, even if the additional packages had nothing to do with the package being built. In particular, there were two packages (`ubuntu-themes` and `gdb`) that required more time to build in our pre-constructed environments. The largest increase was for the `ubuntu-themes` package, taking 28m56s using `pbuilder` and 32m55s with our build environments, for a 14% slowdown. Investigating, we found that the slowdown was from the "scour" utility minimizing SVG files, and while the slowdown in each run of "scour" was small (less than 0.2 seconds each) the fact that it was run on over 4000 SVG files resulted in the slower package build. Digging further into the reason for the slower run, we discovered that the "scour" utility is written in Python, and our larger pre-constructed environment had 111 system-wide Python packages installed, which resulted in all 111 `PKG-INFO` files being read program startup. In contrast, the minimal `pbuilder` environment only had 3 system-wide Python packages installed, resulting in far less overhead when the Python interpreter was started at each run.

6 Related Work

The study of large software systems has been greatly facilitated by the public nature of open-source software, which provides a wealth of data to draw upon. In this paper, we focused on the specific problem of defining small sets of build environments, an area that we believe has not been explored in these terms before. However, work has been that is based in similar problems arising in large software distributions, and in this section we provide a basic summary of key prior work related to analyzing software distributions and dependencies. For example, González-Barahona *et al.* [7] conducted early studies on Linux distributions, examining metrics such as distribution size, package size in terms of files and lines of code, and programming languages used. Subsequently, Galindo *et al.* [5] used Linux distributions as a basis for studying broader concepts such as variability models for software.

As distributions have grown, the complexity of dependencies and conflicts have proven to be significant challenges for package and distribution maintainers, and modeling these relations has been studied formally. In 2006, Mancinelli *et al.* [10] gave an extended graph model that reflects dependencies and conflicts, and discussed dependency closures in ways similar to our work, but with a focus on binary packages and tasks a maintainer must do to accurately define and visualize package relations. In 2009, de Sousa *et al.* [13] created a similar model and studied the properties of the dependency graph, looking at degree connectivity distribution and modularity, among other measures. While many of these works use the Debian distribution due to its popularity among academics, in 2015 Wang *et al.* [17] perform similar graph modeling to visualize package dependencies in Ubuntu 14.04, although they report only looking at a graph with 2,240 vertices which would be a small subset of the total Ubuntu 14.04 packages.

Researchers have also investigated the impact of package changes on dependency and conflict relationships. In 2013, Di Cosmo, Treinen, and Zacchiroli [4] developed a formal model for analyzing update failures, focusing on end-users updating their systems and maintainers defining appropriate relationships. Their work specifically examined the "co-installability" of binary packages in deployed systems, and did not consider build environments. Another study on end-user installability was conducted by Vouillon and Di Cosmo [16], who linked installability tests to the satisfiability (SAT) problem and raised similar NP-completeness concerns to those we have shown in this paper. They also explored graph simplification and compression techniques similar to those described in Section 3.3. In 2015, Claes *et al.* [1] studied package conflicts and broken packages at a fine-grained level, using daily snapshots reflecting ongoing developer work.

Recently, the appearance of language-specific code repositories for developers, such as NPM (for JavaScript), CRAN (for R), and PyPI (for Python), have raised similar dependency challenges, but in a different context [2, 9, 3]. Of particular interest, Decan, Mens, and Claes [2] show that dependency network topology varies between different language ecosystems, and while they did not use operating system distributions it is reasonable to believe that the range of software included in full operating system distributions will be even more diverse.

The work described above all focused mainly on the challenges end users and developers face in managing dependencies and conflicts for operational systems, and do not address build environments or conflicts between source packages. The only work we're aware of that looks specifically at large-scale software building in open source distributions is the work of Nussbaum [11], which describes rebuilding an entire Debian distribution from source packages. Nussbaum was interested in whether the minimal build environments, as defined by the "build dependencies," were sufficient for each source package, and used a large grid computing infrastructure to create each individual build environment and attempt the builds. Our work, focused on re-using build environments, installs far more packages than the minimal set for any package, so we would not be able to address Nussbaum's question of whether the defined dependencies were sufficient. We would be able to address another question tested by Nussbaum,

however, and that is the question of whether updated tool-chains are still capable of completing a build from the provided source packages.

7 Conclusions and Future Work

In this paper, we explored a problem that arose in applied work analyzing a large collection of open source software through static analysis. Our problems is equivalent to designing environments in which many software packages can be compiled, which we call a "build environment," and minimizing the number of distinct build environments leads directly to performance gains whether interested in the build process or in static analysis. We formalized the problem by examining a graph that we construct from environment conflicts, which we call the "conflict graph," and show that there is a a one-to-one correspondence between the problem of minimizing the number of build environments and the problem of minimizing the number of colors required to color the conflict graph.

In our experimental results, we constructed conflict graphs for various Ubuntu Long-Term Support releases, and computed metrics such as graph size and density for these real-world problems. Since coloring is an NP-hard problem, and conflict graphs can be large, we developed ways to simplify the graph and looked at the problem of coloring increasing-size nested subgraphs. After simplifying the graphs, we used an implementation of various graph coloring approximation algorithms to see how few colors we could achieve in practice, and used the colorings to generate sets of build environments for the Ubuntu 20.04 distribution. We were able to construct a set of 11 build environments that were sufficient to build all 30,646 source packages in Ubuntu 20.04, and a set of just 4 environments for building all of the top 1000 "most popular" source packages. We then software for finding max-cliques, since that provides an upper-bound on a graph's chromatic number, and could conclude that in many cases our constructed set of build environments was the smallest possible, while in the remaining cases it was guaranteed to be within one of optimal. Experiments performing builds with the computed build environments showed dramatic decreases in work required, although also exposed some practical difficulties in large-scale batch package building. Referring back to our original motivation, the results here show the value of modeling the problem with graph coloring, providing significant practical improvements over the *ad hoc* approach.

For future directions, we note that our work made some simplifications related to environment conflicts (described in section 2.3) to simplify modeling the problem constraints. A more precise modeling of constraints, particularly of modeling dependency disjunctions ("or-lists") could lead to some improvements. While we used the same or-list prioritization as the standard Debian build tools, resulting in a good approximation of stand-alone build environments, taking a global view and optimizing across or-lists could provide some advantages. For example, the `pkgconf` package management software is a new re-write of the older `pkg-config` software, and is designed to be a drop-in replacement for any software that uses `pkg-config` for building. Since these two packages can't both

be installed in the same environment, this introduces a conflict that can be satisfied by either package. Packages generally include an "or-list" that says that either `pkgconf` or `pkg-config` can be used, but some older packages prioritize the older `pkg-config` rather than the newer replacement. Would having a global preference for the newer `pkgconf` package provide a larger overlap in preferred dependencies, reducing the conflicts and hence the number of build environments required?

A second question that needs more study is the development of a clear objective when dealing with coloring nested subgraphs. Given our success in finding colorings with a small number of colors, we deferred this question by simply optimizing for each nested graph size independently. While this works for our graphs, there may be situations in which this is not sufficient, and beyond the practical issues it is an interesting problem on its own as a general graph theory problem. In our context, we might include some experimental efficiency results to tune our metrics and objectives.

Finally, our experiments showed that in some cases having very large build environments can result in small numbers of build environments, in some situations (such as invoking Python many times with a large number of installed libraries) this resulted in *slower* build time for some packages. Determining if there is a good trade-off between number of build environments and build environment size would be an interesting future direction.

References

1. Claes, M., Mens, T., Di Cosmo, R., Vouillon, J.: A Historical Analysis of Debian Package Incompatibilities. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. pp. 212–223 (May 2015)
2. Decan, A., Mens, T., Claes, M.: On the topology of package dependency networks: a comparison of three programming language ecosystems. In: Proccedings of the 10th European Conference on Software Architecture Workshops (ECSAW). pp. 1–4 (Nov 2016)
3. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering **24**(1), 381–416 (Feb 2019)
4. Di Cosmo, R., Treinen, R., Zacchiroli, S.: Formal Aspects of Free and Open Source Software Components. In: 11th International Symposium on Formal Methods for Components and Objects (FMCO). pp. 216–239 (2013)
5. Galindo, J., Benavides, D., Segura, S.: Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In: Proceeding of the First International Workshop on Automated Configuration and Tailoring of Applications (ACOTA) (2010)
6. Garey, M.R., Johnson, D.S.: Computers and intractability. W.H. Freeman, San Francisco (1979)
7. González-Barahona, J.M., Robles, G., Ortuño-Pérez, M., Rodero-Merino, L., Centeno-González, J., Matellán-Olivera, V., Castro-Barbero, E.: Analyzing the Anatomy of GNU/Linux Distributions: Methodology and Case Studies (Red Hat and Debian). In: Free/Open Source Software Development. pp. 27–58 (2003)

8. Jackson, I., Schwarz, C., Morris, D.A.: Debian policy manual (version 4.6.0.1) (2021), https://www.debian.org/doc/debian-policy/
9. Kikas, R., Gousios, G., Dumas, M., Pfahl, D.: Structure and Evolution of Package Dependency Networks. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). pp. 102–112 (May 2017)
10. Mancinelli, F., Boender, J., di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). pp. 199–208 (Sep 2006)
11. Nussbaum, L.: Rebuilding Debian using distributed computing. In: Proceedings of the 7th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE). pp. 11–16 (Jun 2009)
12. Pattabiraman, B., Patwary, M.M.A., Gebremedhin, A.H., Liao, W.k., Choudhary, A.: Fast algorithms for the maximum clique problem on massive sparse graphs. In: Algorithms and Models for the Web Graph. pp. 156–169 (2013)
13. de Sousa, O.F., de Menezes M.A., Penna, T.: Analysis of the package dependency on Debian GNU/Linux. Journal of Computational Interdisciplinary Sciences **1**(2), 127–133 (2009)
14. Tate, S.R., Yuan, B.: Minimum size build environment sets and graph coloring. In: Proceedings of the 17th International Conference on Software Technologies, ICSOFT 2022, Lisbon, Portugal, July 11-13, 2022. pp. 57–67 (2022)
15. The Ubuntu Web Team: Ubuntu popularity contest (2021), https://popcon.ubuntu.com/
16. Vouillon, J., Cosmo, R.D.: On software component co-installability. ACM Transactions on Software Engineering and Methodology **22**(4), 34:1–34:35 (Oct 2013)
17. Wang, J., Wu, Q., Tan, Y., Xu, J., Sun, X.: A graph method of package dependency analysis on Linux Operating system. In: 2015 4th International Conference on Computer Science and Network Technology (ICCSNT). pp. 412–415 (Dec 2015)