Autom Softw Eng (2016) 23:619–647
DOI 10.1007/s10515-015-0183-5

Identifying and understanding header file hotspots in C/C++ build processes

Shane McIntosh[1] · **Bram Adams**[2] · **Meiyappan Nagappan**[3] · **Ahmed E. Hassan**[4]

Received: 22 July 2014 / Accepted: 10 July 2015 / Published online: 22 July 2015

Abstract Software developers rely on a fast build system to incrementally compile their source code changes and produce modified deliverables for testing and deployment. Header files, which tend to trigger slow rebuild processes, are most problematic if they also change frequently during the development process, and hence, need to be rebuilt often. In this paper, we propose an approach that analyzes the build dependency graph (i.e., the data structure used to determine the minimal list of commands that must be executed when a source code file is modified), and the change history of a software system to pinpoint header file hotspots—header files that change frequently and trigger long rebuild processes. Through a case study on the GLib, PostgreSQL, Qt, and Ruby systems, we show that our approach identifies header file hotspots that, if improved, will provide greater improvement to the total future build cost of a system than just focusing on the files that trigger the slowest rebuild processes, change the most frequently, or are used the most throughout the codebase. Furthermore, regres-

✉ Shane McIntosh
shanemcintosh@acm.org

Bram Adams
bram.adams@polymtl.ca

Meiyappan Nagappan
mei@se.rit.edu

Ahmed E. Hassan
ahmed@cs.queensu.ca

1 Department of Electrical and Computer Engineering, McGill University, Montreal, Canada

2 Lab on Maintenance, Construction, and Intelligence of Software (MCIS), Polytechnique Montréal, Montreal, Canada

3 Department of Software Engineering, Rochester Institute of Technology, Rochester, USA

4 Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Canada

sion models built using architectural and code properties of source files can explain 32–57 % of these hotspots, identifying subsystems that are particularly hotspot-prone and would benefit the most from architectural refinement.

Keywords Build systems · Performance analysis · Mining software repositories

1 Introduction

Build systems specify how source code, libraries, and data files are transformed into deliverables, such as executables that are ready for deployment. Build tools [e.g. `make` (Feldman 1979)] orchestrate thousands of order-dependent commands, such as those that compile and test source code, to ensure that deliverables are rebuilt correctly. Such a build tool needs to be executed every time developers modify source code, and want to test or deploy the new version of the system on their machine. Similarly, continuous integration and release engineering infrastructures on build servers rely on a fast build system to provide a quick feedback loop.

Since large software systems are made up of thousands of files that contain millions of lines of code, executing a full build can be prohibitively expensive, often taking hours, if not days to complete. For example, builds of the Firefox web browser for the Windows operating system take more than 2.5 h on dedicated build machines.[1] Certification builds of a large IBM system take more than 24 h to complete (Hassan and Zhang 2006). In a recent survey of 250 C++ developers, more than 60 % of respondents report that build speeds are a significant issue.[2] Indeed, while developers wait for build tools to execute the set of commands necessary to synchronize source code with deliverables, they are effectively idle (Humble and Farley 2010).

To avoid incurring such a large build performance penalty for each build performed by a developer, build tools such as `make` (Feldman 1979) provide *incremental builds*, i.e., builds that calculate and execute the minimal set of commands necessary to synchronize the built deliverables with any changes made to the source code. Humble and Farley (2010) suggest that incrementally building and testing a change to the source code should take no more than 1.5 min. Developers have even scrutinized 5-min long incremental build processes,[3] calling the process “abysmally slow.”[4]

To assess build performance bottlenecks in the real world, we asked developers of the GLib and PostgreSQL systems to list the files that slowed them down the most when rebuilding them incrementally. While the reported bottlenecks were often header files that many other files depended upon, and thus took took a long time to rebuild, paradoxically, there were other header files that took a longer time to rebuild, but were not pointed out by the developers. Many of these slower header files were not perceived to be build bottlenecks because they rarely changed over time (and hence, rarely needed to be rebuilt incrementally by the developers).

1 http://tbpl.mozilla.org/.

2 http://mathiasdm.com/2014/01/24/a-c-questionnaire-on-build-speed-the-results-are-in/.

3 https://bugs.webkit.org/show_bug.cgi?id=32921.

4 https://bugs.webkit.org/show_bug.cgi?id=33556.

Hence, the frequency of change that a header file undergoes seems to influence how developers perceive build performance issues, even though it has been largely overlooked by existing build optimization approaches. Our prior finding that only 10–25 % of the source files of large systems like Linux and Mozilla change in a typical month (McIntosh et al. 2011) suggests that traditional build profiling techniques may miss the header files that will really make a difference in the build time during day-to-day development. Instead, build optimization effort should be focused on *header file hotspots*, i.e., header files that not only trigger slow rebuild processes, but also require frequent maintenance.

In this paper, we study header file hotspots in four open source systems, making two main contributions:

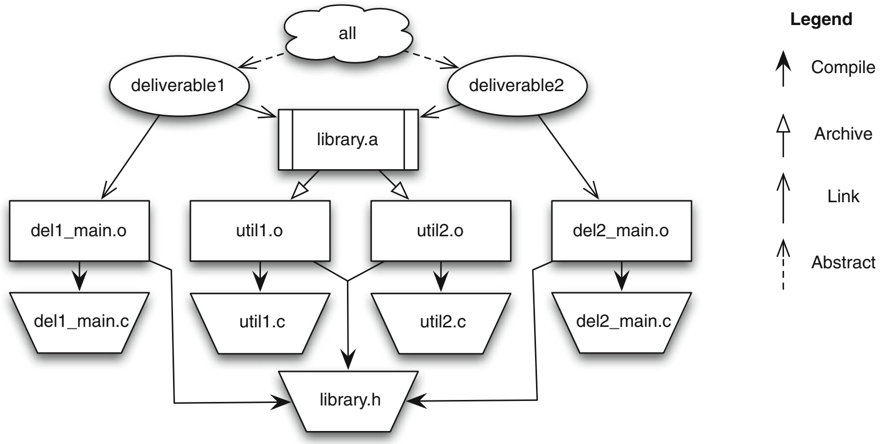
1. We propose an approach to detect hotspots by analyzing the build dependency graph (BDG) and the change history of a system (Sect. 4). We evaluate our approach by simulating the build time improvement of header file hotspots for a developer by using historical data (Sect. 6). We find that optimization of the header files identified by the hotspot approach would lower the total future rebuild cost more than optimization of the header files that trigger the slowest rebuild processes, change the most frequently, or are used the most throughout the codebase.
2. We study the characteristics of header file hotspots in the studied systems (Sect. 7). We find that logistic regression models can explain 32–57 % of the identified header file hotspots using the architectural and code properties of header files. Furthermore, our GLib and Qt models identify hotspot-prone subsystems that would benefit the most from architectural refinement.

The remainder of the paper is organized as follows. Section 2 describes the incremental build process, while Sect. 3 describes how header file hotspots can impact a development team in more detail. Section 4 presents the hotspot detection approach. Section 5 describes the setup of our case study of four open source systems. Section 6 presents the results of our simulation experiment. Section 7 presents the results of our study of the characteristics of header file hotspots. Section 8 discloses the threats to the validity of our study. Section 9 surveys related work. Finally, Sect. 10 draws our conclusions.

2 Incremental builds

The build process for a software system is typically broken down into two main phases (Adams et al. 2008). The first phase is *configuration*, where the build system selects: (1) build tools (e.g. compilers and linkers), and (2) features to include in the build (e.g. Windows vs. Android front-end). The next phase is *construction*, where relevant source code and data files are translated (compiled) into deliverables by orchestrating several order-dependent commands. In addition, deliverables are certified by executing suites of automated tests, and finally bundled with product documentation and data files for delivery to end users.

Developers who make source code changes would like to quickly produce modified deliverables in order to test their changes. Hence, the cornerstone feature of a build



(a) Build dependency graph.

```
CC = gcc
LIBTOOL = libtool

.PHONY: all
all: deliverable1 deliverable2

deliverable1: del1_main.o library.a
        $(CC) -o $@ $^ # recipe 1

deliverable2: del2_main.o library.a
        $(CC) -o $@ $^ # recipe 2

library.a: util1.o util2.o
        $(LIBTOOL) -static -o $@ $^ # recipe 3

%.o: %.c library.h
        $(CC) -c $< # recipe 4
```

(b) make implementation

Fig. 1 An illustrative build dependency graph and its `make` implementation

system is the incremental build, which can reduce the cost of a full build dramatically. After performing a full build that produces initial copies of the necessary deliverables, incremental builds only execute the commands necessary to update the deliverables (“build targets”) impacted by source code changes.

For example, consider the BDG depicted in Fig. 1a, which represents the dependencies in the `make` specification of Fig. 1b. The `all` node in the graph is *phony*, i.e., a node used to group deliverables together into abstract build phases rather than to represent a file in the filesystem. The full build will execute four compilation commands (recipe 4) to produce build targets `del1_main.o`, `util1.o`, `util2.o`, and `del2_main.o`, as well as an archive command (recipe 3) to produce `library.a`, and finally, two link commands (recipes 1 and 2) to produce `deliverable1` and `deliverable2`. If `del1_main.c` is modified after a full build has been performed, an incremental build only needs to recompile `del1_main.o` and re-link `deliverable1`. As software systems (and BDGs) grow, the minimizing behaviour of incremental builds saves developers time.

3 Header file hotspots

Although incremental builds tend to save time, changes to header files often trigger slow rebuild processes (Lakos 1996). For example, Fig. 1a shows that changes to `library.h` will trigger the equivalent of a full build, since all four `.c` files reference `library.h`, and will thus need to be recompiled when it changes. In turn, `library.a` will be re-archived and the two deliverables will be re-linked.

To better understand how developers are impacted by such build performance bottlenecks (e.g. header files that trigger slow rebuild processes), we asked the three most active contributors to GLib and PostgreSQL (two long-lived and rapidly evolving open source systems) to pick five files that they believe slow them down the most when rebuilding. Surprisingly, the files that were reported as bottlenecks were not the ones with the worst raw build performance. In fact, of the bottlenecks reported by the three developers, the files with the worst performance appear 61st (GLib) and 32nd (PostgreSQL) in the lists of files ordered by actual rebuild cost (i.e., the time taken to incrementally build the system after a change to one of those files). Indeed, the respondents seemed to have most of their build performance issues with files that we measured to be relatively fast to rebuild. When asked why they did not select the slower files, one GLib developer responded: "because none of these [files] change often."

At first glance, this insight might seem counterintuitive. However, consider the scenario depicted in Fig. 2 with a header file hotspot and a team of four developers: A, B, C, and D. First, changing the hotspot file impacts the original developer. For example, if developer A modifies H, the change would trigger the slow rebuild process of H in A's copy of the source code. After committing the hotspot change to the version control system, the change to the hotspot impacts other team members as well. When developers B, C, and D update their copies of the source code and receive A's change to H, it will also trigger the slow rebuild process of H on their machines. If H tends to change often, the slow rebuilds on developers' machines keep on repeating, accumulating a large incremental build cost over time.

Based on this insight, this paper analyzes whether the header file hotspots (i.e., header files that not only trigger long rebuild processes, but also tend to change frequently) are better indicators of files that will slow the rebuild process in the future, and hence should be optimized now to save developers time. In order to understand how such reduction of rebuild cost can be achieved, we use logistic regression models to study actionable factors that impact header file hotspot likelihood. Such factors correspond to common source code (e.g. file fan-in) and code layout properties (e.g. the subsystem that a file belongs to). Of course, making fewer changes to the code is not a feasible option for reducing build activity, since after all, the software needs to evolve to implement changing requirements.

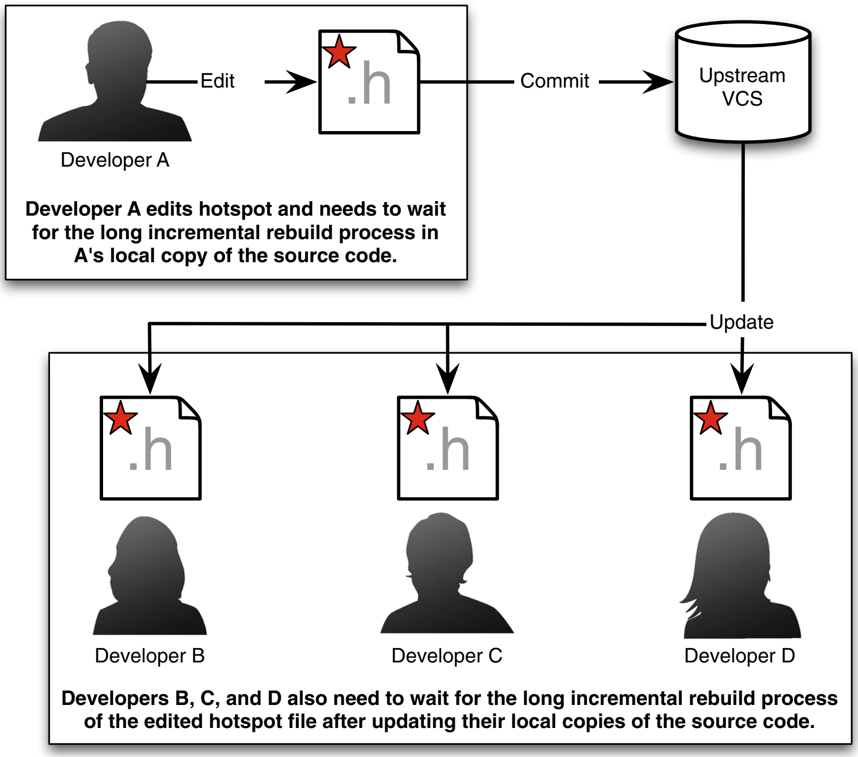

Edit
.h
Commit
Upstream VCS
Developer A
Developer A edits hotspot and needs to wait for the long incremental rebuild process in A's local copy of the source code.
Update
.h
.h
.h
Developer B
Developer C
Developer D
Developers B, C, and D also need to wait for the long incremental rebuild process of the edited hotspot file after updating their local copies of the source code.

Fig. 2 An example scenario of the impact that a header file hotspot can have on a development team

4 Hotspot analysis approach

In order to identify header file hotspots, we analyze the BDG and the change history of a software system. Figure 3 provides an overview of our approach, which is divided into the three steps that are described below. In this section, we describe our approach in abstract terms, while details of the prototype implementation used in our case studies are provided in Sect. 5.

4.1 Dependency graph construction

We first extract the BDG of the main build target of a software system (e.g. `all` in Fig. 1a), which is a directed acyclic graph $BDG = (T, D)$ with the following properties:

- Graph nodes represent build targets $T = T_f \cup T_p$, where T_f is the set of concrete files produced or consumed by the build process, T_p is the set of phony targets in the build process, and $T_f \cap T_p = \emptyset$.
- Directed edges denote dependencies $d(t, t') \in D$ from target t to target t'. A dependency exists between targets t and t' if t must be updated when t' changes. Figure 1a shows an example BDG.

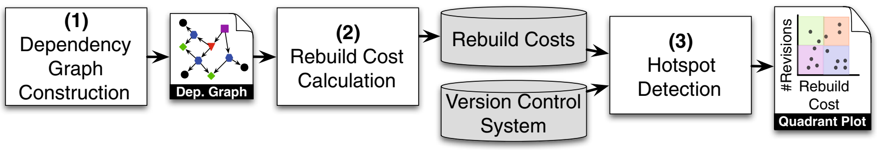


Fig. 3 Overview of our hotspot analysis approach

4.2 Rebuild cost calculation

Although we are mainly interested in header files, our approach can calculate the rebuild cost of each source file in a software system. In order to calculate the rebuild cost of a source file, we build a cost map $CM = (D_r, C)$ with the following properties:

- The set of BDG dependencies $D = D_r \cup D_g$, where D_r is the set of $d(t, t')$ with *recipes* (i.e., build commands that must execute in order to update t when t' changes), D_g is the set of $d(t, t')$ used to order dependencies (i.e., dependencies without recipes), and $D_r \cap D_g = \emptyset$.
- There is a cost $C(d(t, t'))$ associated with each $d(t, t') \in D_r$, which is used to give a weight to each directed edge. This cost may be measured in terms of number of triggered commands, elapsed time, etc.
- CM contains an entry that maps each $d(t, t') \in D_r$ to its cost $C(d(t, t'))$.

The rebuild cost of a source file then is calculated by combining the file's dependencies in the BDG with the edge costs from the CM. The process is split into four steps as described below.

4.2.1 Detect source files

Using the BDG, we detect the set of source files $S = \{s \in T_f \mid |in(s)| > 0 \wedge |out(s)| = 0\}$, where $in(s) = \{d(t, s) \in D\}$ (i.e., dependencies that must be regenerated when s changes) and $out(s) = \{d(s, t) \in D\}$ (i.e., dependencies that regenerate s), and $|X|$ is the cardinality of the set X. In other words, S is the set of non-generated files (no outgoing edges) that are the initial inputs for the main build target.

4.2.2 Detect triggered edges

For each source file node $s \in S$, we identify the set of edges $E(s)$ that will be triggered should s change by selecting all edges that transitively depend on s in the BDG. In other words, we perform a transitive closure of $d(s, t)$ on the BDG, and filter away edges that are not present in the BDG.

4.2.3 Filter duplicate edges

Since the same recipe may be attached to multiple outgoing edges of a given build target t, we count each such recipe only once by filtering out all but one of the corresponding edges $d(t, t')$ from $E(s)$. We apply this filter to all dependencies $d(t, t') \in E(s)$ to obtain $E'(s)$.

For example, Fig. 1a shows that when either `util1.o` or `util2.o` is updated, `library.a` must be re-archived. The `make` implementation in Fig. 1b shows that in such a case, the re-archiving of `library.a` only needs to be performed once. In this case, we would filter the edge between `library.a` and `util2.o` out of $E(s)$ to obtain $E'(s)$.

4.2.4 Aggregate cost of triggered edges

Finally, to calculate the rebuild cost of a source file s, we begin by looking up each edge $d(t, t') \in E'(s)$ in the CM. Any edge that appears in $E'(s)$, but does not appear in CM (e.g. $d(t, t') \in D_g$) is assumed to have no cost. The rebuild cost is then calculated by summing up the costs of the edges in $E'(s)$ that were found in the CM.

4.3 Hotspot detection

Software systems evolve through continual change in the source code, build system, and other artifacts. Changes to files are logged in a version control system (VCS), such as Git. To identify hotspots, we need to calculate the rate of change of each source file, i.e., the number of revisions of the file that are recorded in the VCS, then plot this against the rebuild cost for each file. Similar to Khomh et al. (2011), we divide the plot into four quadrants:

Inactive Files that rarely change and that trigger quick rebuild processes. Optimizing the build for these files is unnecessary.

High churn Files that frequently change, but trigger quick rebuild processes. These files are low-yield build optimization candidates because although they endure heavy maintenance, they do not cost much to rebuild.

Slow build Files that rarely change, but trigger slow rebuild processes. These files are low-yield build optimization candidates.

Hotspot Files that frequently change and trigger slow rebuild processes. These files are high-yield build optimization candidates.

The quadrant thresholds can be dynamically configured to suit the needs of the development team. Initially, thresholds may be selected using intuition, however later on, nonfunctional requirements could specify a maximum rebuild cost according to a system's common rate of file change.

5 Case study setup

We perform a case study on four open source systems in order to: (1) evaluate our header file hotspot detection approach, and (2) study the characteristics of real-world header file hotspots. Hence, our case study is divided into two sections accordingly, which we motivate below:

Table 1 Characteristics of the studied systems

	GLib	PostgreSQL	Qt	Ruby
Domain build technology	Development library autotools	DBMS Autoconf, `make`	UI framework QMake	Programming language autoconf, `make`
Version	2.36.0	9.2.4	5.0.2	1.9.3
Release date	2013-03-25	2013-04-04	2013-07-03	2011-10-31
System size (kSLOC)	401	658	5132	1098
# BDG nodes	3375	4637	38,235	1560
# BDG edges	121,710	59,676	2,752,226	6240

For each studied system, we extract 2 years of historical data just prior to the release dates

Evaluation of our hotspot detection approach (*Sect.* 6) Since rebuild cost, rate of change, and dependencies on other files individually can also be used to prioritize files for build optimization, we want to evaluate whether the hotspot heuristic truly identifies the most costly header files.

Analysis of header file hotspot characteristics (*Sect.* 7) Since code changes are required to address defects or add new features, one cannot simply avoid changing the code. Instead, build optimization effort must focus on controllable (actionable) properties that influence header file hotspot likelihood. Hence, we set out to study the relationship between controllable header file properties and hotspot likelihood.

The remainder of this section introduces the studied systems, provides additional detail about our implementation of the hotspot detection approach proposed in Sect. 4, and compares the build performance of header files to other files in the studied systems.

5.1 Studied systems

We select four long-lived, rapidly evolving open source systems in order to perform our case study. We select systems of different sizes and domains to combat potential bias in our conclusions. Table 1 provides an overview of the studied systems.

GLib is a core library used in several GNOME applications.[5] *PostgreSQL* is an object-relational database system.[6] *Qt* is a cross-platform application and user interface framework whose development is supported by the Digia corporation, however welcomes contributions from the community-at-large.[7] Ruby is an open source programming language.[8]

The studied systems use different build technologies (e.g. GNU Autotools and QMake). However, each studied build technology eventually generates `make` specifications from higher level build specifications. The choice of studying `make`-based

[5] https://developer.gnome.org/glib/.

[6] http://www.postgresql.org/.

[7] http://qt.digia.com/.

[8] https://www.ruby-lang.org/.

build systems is not a coincidence, since such build systems are the de facto standard for C/C++-based software projects (McIntosh et al. 2015), which are the projects that typically use header files.

5.2 Implementation details

5.2.1 Dependency graph construction and rebuild cost calculation

We first perform a full build of each studied system on the Linux x64 platform with GNU `make` tracing enabled to generate the necessary trace logs. Such a trace log carefully records all of the decisions made by the build tool (e.g. is input file X newer than output file Y?). The generated trace is then fed to the MAKAO tool (Adams et al. 2007), which parses it to produce the BDG and CM. Finally, we implemented the four steps of Sect. 4.2 in a script and applied it to the BDG and CM to calculate the rebuild cost of each source code file $s \in S$.

5.2.2 Edge weight metric

To give the edge weighing function $C(d(t, t'))$ a meaningful concrete value, we use *elapsed time*, i.e., the time spent executing build recipes. For this, we measure the time consumed by each recipe during a full build by instrumenting the shell spawned by the build tool for each recipe's execution. Since varying load on our experimental machines may influence the elapsed time measurements, we repeated the full build process (from scratch) ten times and select the median elapsed time for each recipe.

After ten repetitions, we find that the standard deviation of the elapsed time for any given command does not exceed 0.5 s and the median standard deviation among the ten repetitions does not exceed 0.02 s. Thus, the variability in the elapsed time consumed by a recipe will not substantially skew our results.

5.2.3 Quadrant threshold selection

For the purposes of our case study, we use 90 s as the threshold for rebuild cost, since Humble and Farley suggest this as an upper-bound on the time spent on an incremental build (Humble and Farley 2010). For the rate of change threshold, we select the median number of revisions across all files of a system. Furthermore, to reduce the impact that outliers have on the quadrant plots, we apply the logarithm on both rebuild cost and rate of change values. We normalize rebuild cost and rate of change by dividing each logarithmic value by the maximum so that the quadrant plots of different systems can be compared.

5.3 Preliminary analysis of header file build performance

Prior to performing our case studies, we first perform a preliminary analysis to evaluate whether header files are truly the source of the most problematic build hotspots in the studied systems. Indeed, while prior work has focused on header file optimization (Dayani-Fard et al. 2005; Yu et al. 2003, 2005), it is unclear whether they are truly the largest source of build hotspots. Since header files represent interfaces (which ought to be more stable over time), they may not necessarily change as frequently as regular source code files. It is conceivable that core implementation files that change often and generate a substantial amount of link-time build activity may also be hotspots that are worthy of optimization effort (Lakos 1996).

5.3.1 Approach

Figure 4 plots the rebuild cost of each source file $s \in S$ in increasing order for each of the studied systems. The figures in the column on the left show the rebuild costs of header files, while the figures in the column on the right show the rebuild costs of the other source files in each of the studied systems. In addition, we show quadrant plots of the rebuild costs versus the number of revisions of each source file $s \in S$ in Fig. 5.

5.3.2 Results

Figure 4 shows that, as expected, almost all header files trigger longer rebuild processes than other file types do. This is primarily because when a header file is changed, all files that `#include` it must be recompiled (*cf.* Sect. 2). The majority of GLib header files trigger rebuild processes of more than 60 s (Fig. 4a). Several Qt header files trigger rebuild processes of more than 15 min (900 s), with extreme cases reaching over 2 h (Fig. 4e). In all of the studied systems, the median rebuild cost for header files is at least 10 times larger than the median rebuild cost for the other types of files. Our findings support the argument of Yu et al. (2003), that (false) dependencies in header files can indeed substantially slow down the build process.

Interestingly, header files are not the only source of spikes in rebuild cost. Figure 4b, f, and h show that a small set of other files can trigger rebuild processes of several seconds. Many of these files are `.c` files, for which one would normally expect that several subsequent linker commands may be triggered by updating the object code, however only one compile command should be triggered.

Deeper inspection of the GLib system shows that 89 `.c` files in GLib in fact trigger multiple compile commands. We found that 1 of the 89 `.c` files is imported through the preprocessor into several `.c` files, similar to a header file. Hence, changes to the imported `.c` file trigger compile commands for each `.c` file that includes the file. Another 4 of the 89 multi-compiling `.c` files contain test code that is linked into several test executables. However, each test binary requires the object code of the common files to be generated with different compiler flag settings, which means that the same `.c` file must be compiled once for each compiler flag setting. The remaining 84 of the 89 multi-compiling `.c` files are used to implement a source code generator that produces code that is linked to several test executables. When any of the code generator source files are changed, the tool must be rebuilt, then the generated code must be reproduced, recompiled, and re-linked to the test executables. The GLib code generator is an example of a “build code robot”, as was identified for GCC by Tu and Godfrey (2001).

Figure 4g and h show that the Ruby project has no file that exceeds the 90-s threshold that we selected for header file hotspots. This is likely due to the size of the system

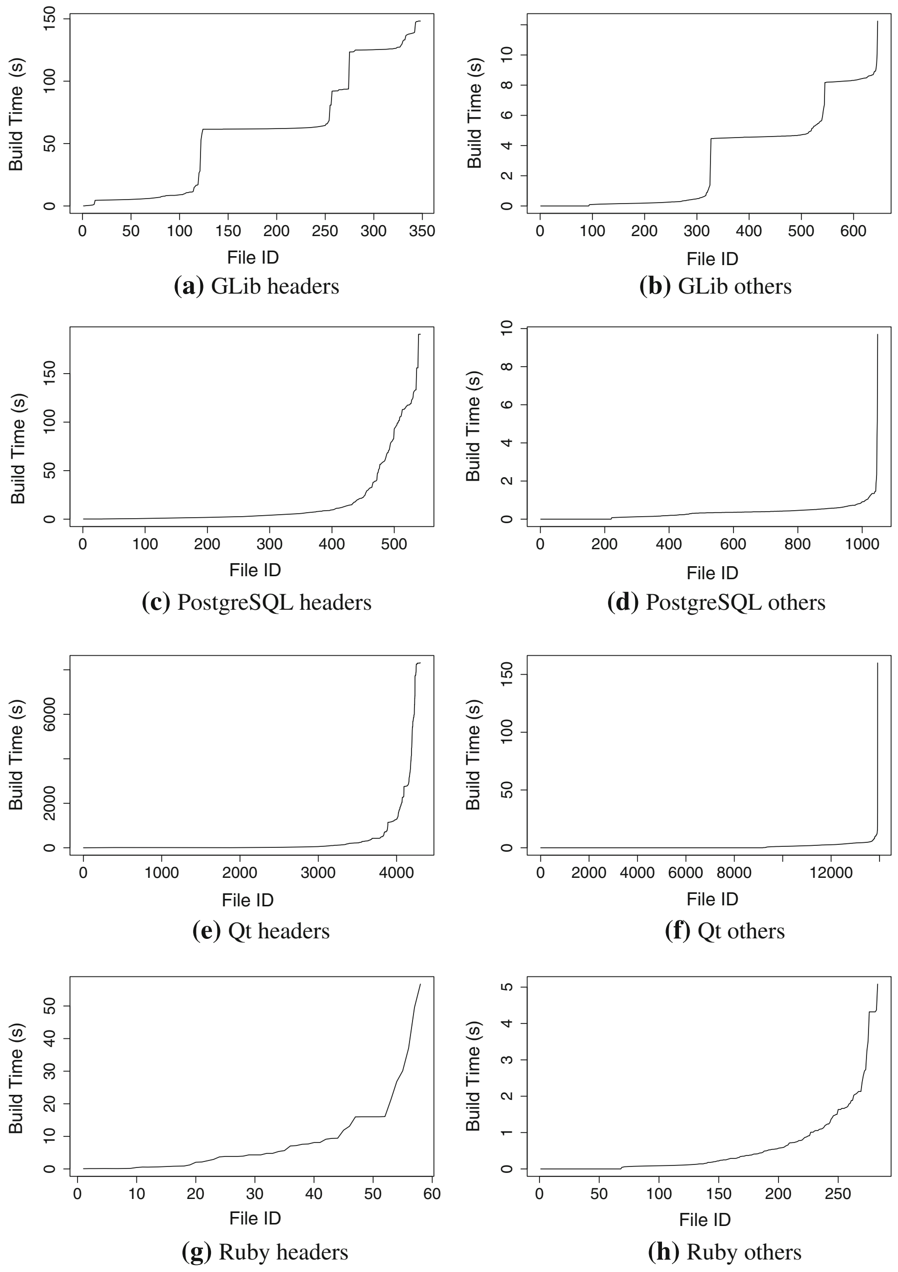


Fig. 4 The rebuild cost of the header and other (primarily source) files in the studied systems

and its BDG, which, as shown in Table 1, has almost an order of magnitude fewer edges than the next smallest system (PostgreSQL). Although Ruby may be free of 90-s header file hotspots, developers of such a small system may be accustomed to a very quick rebuild cycle, and may have a lower threshold for frustration. In a study of

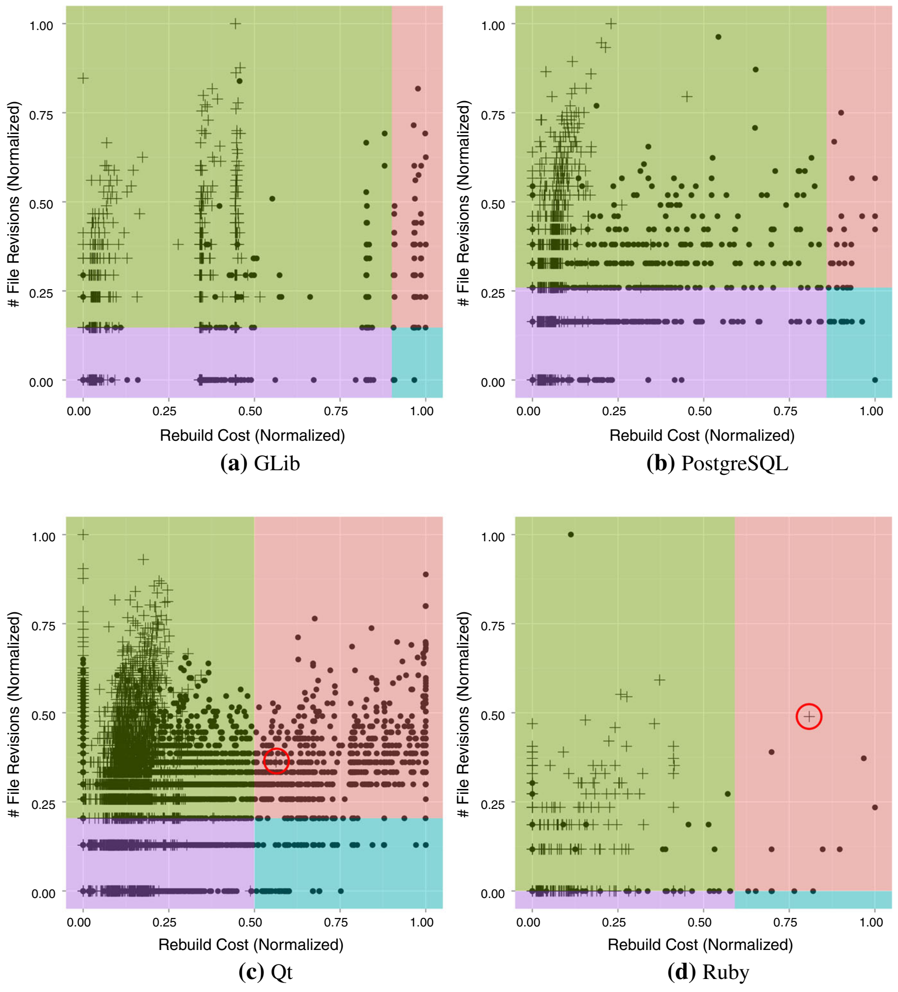


Fig. 5 Quadrant plot of rate of change and rebuild cost. Hotspots are shown in the *top-right* (*red*) quadrant. The *shaded circles* indicate header files, while *plus* (+) symbols indicate non-header files. Non-header file hotspots are *circled in red* (Color figure online)

time delay, Fischer et al. (2005) find that user satisfaction degrades linearly as delay increases from 0 to 10 s. We, therefore, set the threshold for Ruby hotspots to 10 s for the remainder of the paper.

Non-header files do not generate enough build activity to be of concern for hotspot detection. Figure 5 shows the source files that land in each of the four quadrants. Header files are plotted using shaded circles, while other files are plotted using plus (+) signs. Files that land on quadrant borders are conservatively mapped to the lower quadrant.

The quadrant plots in Fig. 5 show that only two non-header files appear in the hotspot quadrant. The first is a Bison grammar file `parse.y` in the Ruby system. Changing the grammar file causes both `parse.c` and `parse.h` to be regenerated, which in turn triggers several recompilations. The second non-header file hotspot is a Qt file `qtdeclarative/tests/auto/shared/util.cpp`, which contains the testing utility code that causes several test binaries to be re-linked. Although each change to the test utility implementation triggers a rebuild process of 159 s, Fig. 4e shows that there are several Qt header files that, when they change, trigger rebuild processes that take hours.

Although some implementation files take a long time to rebuild, the median rebuild cost for header files is at least ten times larger than the median rebuild cost for the other types of files. For this reason, while our hotspot detection approach is generic enough to be applied to any source file, the remainder of this paper focuses on header file hotspots.

6 Case study 1: evaluation of the hotspot detection approach

While our quadrant plots can identify header file hotspots, it is not clear whether build optimization effort that is focused on such hotspots would yield a substantially larger reduction in future build cost than other hotspot detection approaches. In this section, we discuss a case study that we have performed in order to evaluate our hotspot heuristic.

6.1 Approach

In order to evaluate our hotspot heuristic, we compare its decrease in future rebuild cost when prioritizing files for build optimization to the decrease in future rebuild cost when using heuristics based on the following:

Individual rebuild cost Header files that trigger slow rebuild processes are likely to be costly.
Rate of change Header files that are changing frequently are likely to be costly.
File fan-in Prior header file optimization approaches focus on header files that have the highest file fan-in (Yu et al. 2003, 2005), i.e., number of modules that use the functionality defined or declared in the header. These approaches implicitly assume that header files with the highest file fan-in trigger the most build activity.

In order to perform this comparison, we perform a simulation exercise using 2 years of historical data from each studied system, which we divide into training and testing corpora. Figure 6 provides an overview of the steps in our simulation exercise. We describe each step in the simulation below.

6.1.1 Extract historical data

We allocate the year of historical data just prior to the studied releases shown in Table 1 to the testing corpus. Then, we allocate the year prior to the testing corpus to the training corpus. We do so because we want to evaluate the approaches on a time

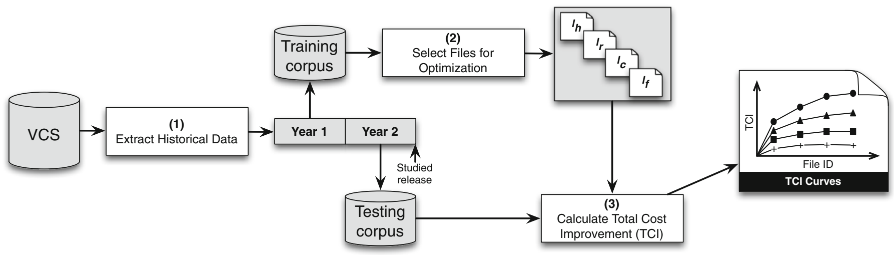

VCS
(1) Extract Historical Data
Year 1
Year 2
Training corpus
(2) Select Files for Optimization
l_h
l_r
l_c
l_f
Studied release
Testing corpus
(3) Calculate Total Cost Improvement (TCI)
TCI
File ID
TCI Curves

Fig. 6 Overview of our simulation exercise

period where we are sure that developers would be rebuilding the system frequently. The period leading up to a release is guaranteed to require frequent rebuilds due to active development.

The build cost and build changes are calculated differently. Build change is measured twice—once in the training corpus, and again in the testing corpus. This makes the data representative for changes in each corpus. On the other hand, we measure rebuild cost on the actual release instead of on an intermediate version released in between the training and testing period. We do so out of convenience, since measuring rebuild cost and extracting the build dependency graph requires a buildable version of the whole system, whereas intermediate versions more often than not are broken in several subsystems (especially in large systems like Qt). Although our evaluation hence combines rebuild cost measurements with change data that is 1 year older, in practice the mismatch is quite limited, as shown by our results.

6.1.2 Select files for optimization

For each of the four approaches, we identify the top N header files that should be optimized for build speed based on analysis of the training corpus. Depending on the heuristic, the top N corresponds to the header files that occupy the hotspot quadrant (l_h), or the N files with the highest individual rebuild cost (l_r), rate of change (l_c), or file fan-in (l_f).

6.1.3 Calculate total cost improvement (TCI)

To evaluate the impact of improving the top N header files in the testing corpus suggested by a particular heuristic calculated in the training corpus, we calculate the *Total Cost Improvement* (TCI). This is the percentage of reduction in the *Total Rebuild Cost* (TRC, i.e., the sum of the rebuild cost of all header files h modified across the changes in the testing corpus) that would be achieved when replaying all required builds of the testing corpus if the individual rebuild costs of each header file in l_h, l_r, l_c, or l_f (i.e., the top N header files of each of the four approaches) were reduced by 10 %[9] prior to entering the testing corpus.

[9] The simulation was repeated for 20 and 50 % improvements, which yielded similar results.

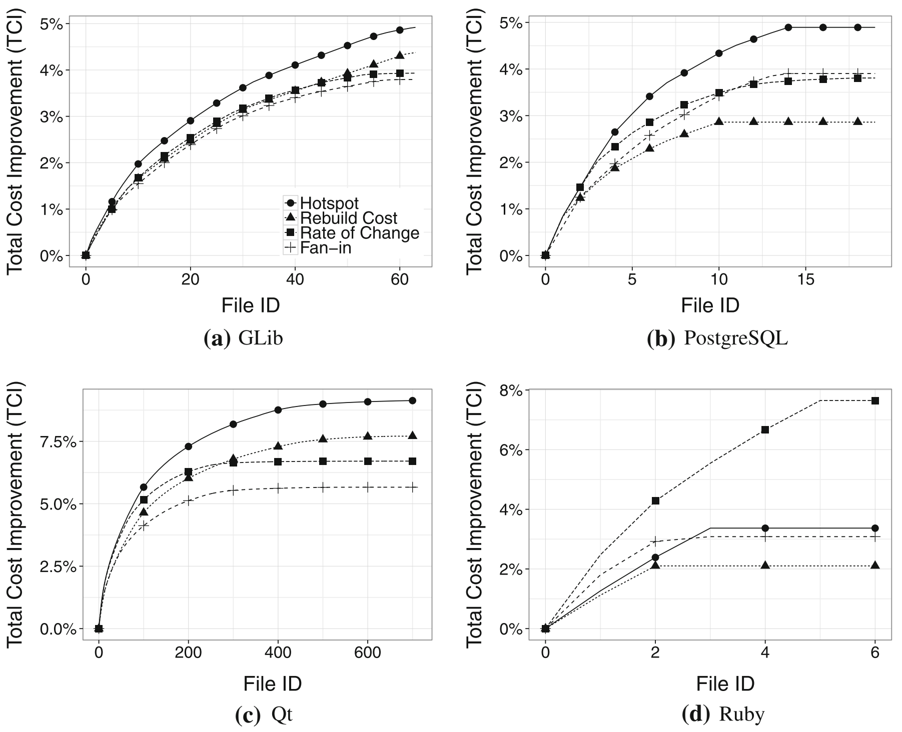


Fig. 7 Cumulative curves comparing the four approaches for selecting header files for build performance optimization. The total cost improvement (TCI) measures the reduction of time spent rebuilding in the future (testing corpus) when the performance of the selected header files are improved by 10 %

For example, we first calculate the total rebuild cost in the testing corpus, and then, if a particular heuristic suggests that we optimize header files A, B, and C based on the training corpus, we recalculate the hypothetical total rebuild cost assuming that the individual rebuild costs of header files A, B, and C were reduced by 10 %. Then, the TCI for the heuristic is calculated as: $TCI = \frac{TRC_{\text{actual}} - TRC_{\text{hypothetical}}}{TRC_{\text{actual}}}$. Note that in this paper, we consider a reduction of 10 % in individual build cost, independent of specific approaches (e.g. refactorings) that can be used to obtain a 10 % reduction (Dayani-Fard et al. 2005).

Figure 7 compares the TCI of l_h, l_r, l_c, and l_f for each of the studied systems. These curves are cumulative, meaning that, for instance, the TCI shown for file #2 in each curve is the TCI in which the top two files (#1 and #2) have been improved. The maximum number N for which we calculate TCI corresponds to the total number of header files in the hotspot quadrant of l_h.

6.2 Results

The TCI of the hotspot heuristic exceeds the TCI of files with the highest individual rebuild cost, rate of change, or file fan-in. Indeed, Fig. 7 shows that the TCI of l_h

exceeds those of l_r, l_c and l_f in three of the four studied systems. In the Ruby system, simply ordering files by their change frequency yields the highest TCI. Individual rebuild cost does not help to select the files that will yield a high TCI because most Ruby header files rebuild quickly (*cf.* Sect. 5).

We performed a Kruskal–Wallis rank sum test to check whether there is a statistically significant difference between the TCI values of l_h, l_r, l_f and l_c ($\alpha = 0.05$). The test results for the GLib, PostgreSQL, and Qt systems show that the differences in TCI are significant ($p \ll 0.05$), indicating that the hotspot analysis selects more costly header files than just considering the file fan-in, the individual rebuild cost, or the rate of change of a header file separately. In the case of Ruby, a Wilcoxon signed rank test indicates that the hotspot analysis (l_h) performs significantly worse than the rate of change (l_c).

Figure 7 shows that Qt achieves the largest TCI rates, with l_h reaching a peak of 9.3 % and l_r reaching 8.7 %. Note that TCI values in this simulation are theoretically constrained to a maximum of 10 %, since this is the amount that the individual rebuild cost of the selected header files were improved by. Moreover, TCI values of 8.7 and 9.3 % equate to a total rebuild cost savings of 8.4 and 8.9 days respectively—on average, a savings of 3.3 and 3.7 min of rebuild cost per change. GLib and PostgreSQL both achieve maximum TCI values of 4.9 %, which equate to a savings of 49.0 and 7.4 min, or 4.0 and 1.4 s per change respectively. Although Ruby achieves a high TCI value of 7.6 % by optimizing the most frequently changing files, due to the rapid speed of the Ruby build process, this only equates to a total savings of 38.7 or 0.16 s per change. Large and complex systems like Qt can really lower rebuild costs with carefully focused build optimization.

The hotspot heuristic tends to select more costly header files that yield a higher total cost improvement than other header file selection heuristics, especially in larger systems with more complex build dependency graphs.

6.3 Discussion

It is important to note that TCI is an under-approximation for the true total rebuild cost for two reasons. First, TCI assumes that each change will only be rebuilt once, when in reality, a build will run several times by the developer making the change, and by the other developers of the system (*cf.* Sect. 2). Since the number of times a build was executed for a particular change is typically not recorded, we assume the minimum case (i.e., just once). Second, TCI assumes that rebuild cost improvements to header files are independent, i.e., by improving the rebuild cost of a header file A, we do not improve the rebuild cost of another header file B in our simulation. In reality, due to overlapping dependencies, this assumption likely will not hold. In both cases, our reported TCI values correspond to a lower bound of the actual TCI—the actual cost savings will be higher in practice.

7 Case study 2: hotspot characteristic analysis

To help practitioners avoid creating header file hotspots or find opportunities for refactoring, we analyze whether header file properties could give concrete indications of hotspot likelihood. To do so, we build logistic regression models and measure the effect that each property has on hotspot likelihood.

When selecting explanatory variables for our models, we discarded change frequency, since it is not an actionable factor, i.e., changes that fix defects and add features cannot be avoided. Instead, build optimization effort must focus on reducing the rebuild cost of a header file by either: (1) shrinking the set of triggered edges $E'(s)$, or (2) reducing the complexity of the header file itself (to reduce its individual compilation time). The latter suggests that header file size and complexity metrics should be added to our models, while the former suggests that we should consider code layout (i.e., architecture) as well.

7.1 Approach

We build *logistic regression models* to check for a relationship between header file properties and hotspot likelihood. A logistic regression model will predict a binary outcome variable (whether or not a header file appears in the hotspot quadrant of Fig. 5) based on the values of a given set of explanatory variables.

Table 2 lists the code content and layout properties that we considered, and provides our rationale for selecting them. Each metric is measured using the released versions of the studied systems presented in Table 1. Code layout metrics are derived from the pathname of each source file. Directory and file fan-in are calculated based on code dependency information extracted using the Understand static code analysis tool.[10] Source lines of code are counted using the SLOCCount tool.[11] Number of includes is calculated using common UNIX tools to select `#include` lines that refer to files within each software system.

Since our goal is to understand the relationship between the explanatory variables (code layout and content properties) and the dependent variable (hotspot likelihood), which is similar to prior work of Mockus (2010) and others (Cataldo et al. 2009; Shihab et al. 2010), we adopt a similar model construction approach. Moreover, since we do not intend to use the models for prediction, but rather for explanation, we do not split the datasets into training and testing corpora as was done in Sect. 6, but rather build models using both years of the historical data together. To lessen the impact of skew, we log-transform the SLOC, number of includes, depth, and file and directory fan-in variables. We build models for each studied system separately.

7.1.1 Data preparation and model construction

We check for variables that are highly correlated with one another prior to building our models, and also check for variables that introduce multicollinearity into preliminary

[10] http://www.scitools.com/index.php.

[11] http://www.dwheeler.com/sloccount/.

Table 2 Source code properties used to build logistic regression models that explain header file hotspot likelihood

Property	Description	Rationale
Code layout		
Subsystem	The top-level directory in the path of a header file	Certain subsystems have a more central role and thus may be more susceptible to header file hotspots
Depth	The number of subdirectories in the header file's path relative to the top directory of the system	Since header files that appear in deeper directory paths are likely more specialized and hence impact fewer deliverables than header files at shallower directory paths, they likely have a smaller impact on the build process, and are thus less likely to be hotspots
Directory fan-in	The number of other directories whose source files refer to code within this header file	Header files with code dependencies that are more broadly used throughout the codebase are likely to have a higher rebuild cost, and thus are more likely to be hotspots
Code content		
File fan-in	The number of other source files referring to code within this header file	The more source files that rely on a header file, the more likely it is to be a hotspot
Source lines of code	The number of non-whitespace, non-comment lines	Larger header files are more likely hotspots
Number of includes	The number of distinct imported interface files, i.e., `#include` statements	Yu *et al.* suggest that including several interface files bloats the build process (Yu et al. 2003). Hence, we suspect that importing many other header files will increase hotspot likelihood

models. We use Spearman rank correlation (ρ) to check for highly correlated variables instead of other types of correlation (e.g. Pearson) because rank correlation does not require normally distributed data. After constructing preliminary models, we check them for multicollinearity using the variance inflation factor (VIF) score. A VIF score is calculated for each explanatory variable used by a preliminary model. A VIF score of one indicates that no collinearity is introduced by the variable, while values greater than one indicate the ratio of inflation in the variance explained due to collinearity. As suggested by Fox (2008), we select a VIF score threshold of five. Neither correlation nor VIF analysis identified any variables that are problematic for our models ($|\rho| \geq 0.6$ or $VIF \geq 5$).

Finally, to decide whether an explanatory variable is a significant contributor to the fit of our models, we perform drop-one tests (Chambers and Hastie 1992) using the implementation provided by the core `stats` package of R (R Core Team 2013). The test measures the impact of an explanatory variable on the model by measuring the *deviance explained* (i.e., the percentage of deviance that the model covers) of models consisting of: (1) all explanatory variables (the full model), and (2) all explanatory variables except for the one under test (the dropped model). A χ^2 likelihood ratio test is applied to the resulting values to indicate whether each explanatory variable improves the deviance explained by the full model to a statistically significant degree ($\alpha = 0.05$). We discard those variables that do not improve the deviance explained by a significant amount.

7.1.2 Model analysis

In order to compare the effect that each statistically significant header file property has on hotspot likelihood, we extend the approach of Cataldo et al. (2009). First, a typical header file is imitated by setting all explanatory numeric variables to their median values and categorical/binary values to their mode (most frequently occurring) category. The model is then applied to the typical header file to calculate its *predicted probability*, i.e., the likelihood that the typical header file is a hotspot, which we call the standard median model (SMM). One by one, we then modify each explanatory variable of the typical header file in one of two ways:

Numeric variable We add one standard deviation to the median value and recalculate the predicted probability.
Categorical/binary variable The predicted probability is recalculated for each category except for the mode.

The recalculated predicted probabilities are referred to as the increased median model (IMM) values. Note that the SMM is a fixed value while IMM is calculated for each explanatory variable. The Effect Size ES of an explanatory variable X is then calculated as:

$$ES(X) = \frac{IMM(X) - SMM}{SMM} \tag{1}$$

Variables can have positive or negative *ES* values indicating an increase or decrease in hotspot likelihood relative to SMM respectively. The farther the value of $ES(X)$ is from zero, the larger the impact that X has on our models. For example, an ES value of 0.51 means that the IMM is 51 % higher than the SMM.

7.2 Results

Tables 3 and 4 shows that our complete models achieve between 32 % (Ruby) and 57 % (GLib) deviance explained. Our models could be likely improved by adding additional header file properties, or even properties extracted from the build system. However, we believe that these models provide a sound starting point for explaining header file hotspot likelihood.

Tables 3 and 4 also shows the change in deviance explained reported by the drop-one test (ΔDE) for each variable. The ΔDE measures the percentage of the deviance explained by the model that can only be explained by a particular variable. Since multiple variables may explain the same header file hotspots, the ΔDE values do not sum up to the total deviance explained by the full model.

Table 3 Logistic regression model statistics for the larger studied systems (i.e., GLib and Qt)

GLib				Qt			
Deviance explained		57 %		Deviance explained		49 %	
Metric	Subdir.	ΔDE (%)	$ES(X)$	Metric	Subdir.	ΔDE (%)	$ES(X)$
Subsystem	gio	28***	†	Subsystem	qtwebkit	23***	†
	tests		2.91		qtimageformats		−0.69
	gmodule		4.73		qtactiveqt		−0.52
	build		49.94		qtsvg		−0.07
	./		158.62		qtdoc		0.43
	gobject		>1,000		qtgraphicaleffects		0.43
	glib		>1,000		qtscript (+7 others)		>1,000
Depth		◇		Depth		5***	−0.58
Directory fan-in		1*	4.11	Directory fan-in		1***	0.98
File fan-in		5***	1.10	File fan-in		13***	2.53
SLOC		1*	0.39	SLOC		1***	0.44
Includes		◇		Includes		1***	−1.0

Deviance explained (DE) indicates how well the model explains the build hotspot data. ΔDE measures the impact of dropping a variable from the model, while $ES(X)$ measures the effect size (see Eq. 1), i.e., the impact of explanatory variables on model prediction

† Mode values of categorical variables are part of the SMM calculation and hence cannot be calculated using IMM

Statistical significance of explanatory power according to Drop One analysis: ◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

Table 4 Logistic regression model statistics for the smaller studied systems (i.e., PostgreSQL and Ruby)

	PostgreSQL		Ruby	
Deviance explained	47 %		32 %	
Metric	ΔDE	$ES(X)$	ΔDE	$ES(X)$
Subsystem	◇		◇	
Depth	◇		◇	
Directory fan-in	◇		◇	
File fan-in	47 %***	8.75	32 %***	3.38
SLOC	◇		◇	
Includes	◇		◇	

Deviance explained (DE) indicates how well the model explains the build hotspot data. ΔDE measures the impact of dropping a variable from the model, while $ES(X)$ measures the effect size (see Eq. 1), i.e., the impact of explanatory variables on model prediction

Statistical significance of explanatory power according to Drop One analysis: ◇ $p \geq 0.05$; * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$

In the larger studied systems, header file hotspots are more closely related to code layout than to the content of a file. Table 3 shows that there is a drop in deviance explained of 28 and 23 % in the GLib and Qt systems respectively when the subsystem explanatory variable is excluded from the model. Furthermore, although the impact is small, directory fan-in explains a statistically significant amount of deviance in the GLib and Qt systems. On the other hand, in those systems, file size and number of includes offer little explanatory power, and although GLib and Qt models without file fan-in drop in explanatory power by 5 and 13 % respectively, the decrease is smaller than that of the subsystem variable. Hence, most of the explanatory power of the GLib and Qt models is derived from code layout properties, such as the subsystem of a file.

Filesystem depth provides additional explanatory power (5 %) in Qt, which is the largest studied system. The negative effect size indicates that as we move deeper into the filesystem hierarchy, hotspot likelihood decreases, suggesting that more central header files (located at shallower filesystem locations) are more prone to build performance issues.

On the other hand, for the smaller systems, code layout properties offer little hotspot explanatory power. Table 4 shows that the subsystem variable does not contribute a significant amount of explanatory power to the PostgreSQL and Ruby models. Moreover, despite the fact that 93 % of the PostgreSQL hotspots reside in the `include` subsystem, this corresponds to 79 % of all PostgreSQL header files, making this a less distinguishing variable.

Furthermore, filesystem depth is not a significant contributor to our PostgreSQL and Ruby models. The vast majority of hotspots in the PostgreSQL and Ruby systems are found in their `include` and top directory subsystems respectively, which do not have complex subdirectory structures within them. On the other hand, hotspots are more evenly distributed among subsystems in the Qt system, and hence the depth metric provides additional explanatory power there.

File fan-in provides all of the explanatory power in the smaller studied systems. Although file fan-in provides a significant amount of explanatory power to all of the models, file fan-in impacts the performance of the smaller PostgreSQL and Ruby system models the most. File fan-in calculates the source files that are directly depending on the functionality provided within the header file. In this sense, file fan-in provides an optimistic (minimal) perspective of dependencies among code files. In smaller systems, this optimistic perspective is sufficient, whereas in larger systems with many subsystems (and a complex interplay between them) where architectural decay has introduced false dependencies among files (Yu et al. 2003), file fan-in no longer accurately estimates the actual set of build dependencies. Spearman rank correlation tests indicate that there is a stronger correlation between the file fan-in and individual rebuild cost of header files in the smaller studied systems ($\rho_{PostgreSQL} = 0.87$, $\rho_{Ruby} = 0.62$) than in the larger ones ($\rho_{Qt} = 0.28$, $\rho_{GLib} = 0.48$). Indeed, in larger systems with a more complex interplay between system components, most of the files including a header file will likely be other header files. Since this additional layer of header file indirection introduces an additional set of unpredictable, but necessary compile dependencies, file fan-in by itself tends to lose its hotspot explanatory power as systems grow.

Yet, even in the smaller PostgreSQL and Ruby systems where file fan-in provides all of the explanatory power, it does not provide a highly accurate estimate of hotspot likelihood. The reason for this discrepancy is twofold. First, while file fan-in provides an optimistic estimate of the compile commands that would be required should the header file change, it can offer little information about the link commands that would be required. For example, two header files with an identical amount of file fan-in may trigger very different amounts of link activity, which would greatly impact the rebuild cost of the header files. Second, being a metric that is calculated based at the code level, file fan-in cannot estimate how frequently a header file will change.

Our models explain 32–57% of identified hotspots. Architectural properties offer much of the explanatory power in the larger systems, suggesting that as systems grow, header file hotspots have more to do with code layout than with code content properties like file fan-in.

7.3 Discussion

The explanatory power of the code layout metrics indicates that there are areas of the larger studied systems that are especially susceptible to header file hotspots (Table 3). Although our regression models seem to suggest that one should simply redistribute header files from subsystems with high hotspot likelihood to the subsystems with lower hotspot likelihood, such a course of action is impractical. Instead, our results should be interpreted as pinpointing the problematic subsystems that would benefit most from architectural refinement, such as a focused refactoring impetus. For example, one could identify the three most hotspot-prone subsystems, and split them into multiple components, or apply the build refactorings of Yu et al. (2003, 2005) or Morgenthaler et al. (2012). Moreover, the impact that code metrics have on our regression models suggests that optimization of the header files in the hotspot-prone subsystems will yield better results if the focus of such optimization is on the reduction of file fan-in, rather than of header file size or the number of includes.

8 Limitations and threats to validity

In this section, we discuss the limitations and the threats to the validity of our study.

8.1 Limitations

Our approach focuses on the detection and prioritization of header file hotspots, but does not suggest automatic hotspot refactorings. In this respect, our approach is similar to defect prediction, which is used to focus quality assurance effort on the most defect-prone modules. Furthermore, automatically proposing fixes for hotspots requires domain-specific expertise. For example, an automatically generated build dependency graph refactoring may fix hotspots in theory, but in practice may require an infeasibly complex restructuring of the system, reducing other desirable properties of a software system like understandability and maintainability. Further work is needed to find a balance between these forces.

An experienced developer may have an intuition about which header files are hotspots, but such a view would be coloured by his or her individual perspective. Moreover, our hotspot detection approach provides automated support to ground developers' intuition with data and through routine (re)application of our approach, a development team can monitor improvement or deterioration of hotspots over time.

8.2 Construct validity

Since build systems evolve (Adams et al. 2008; McIntosh et al. 2012), the BDG itself will change as a software system ages, which may cause the rebuild cost of each file to fluctuate. For the sake of simplicity, our simulation experiment in Sect. 6 projects a constant build cost for each change. Nonetheless, our technique is lightweight enough to recalculate rebuild costs after each change to the build system.

8.3 Internal validity

Since one can only execute a build system for a concrete configuration, we only study a single configuration for each studied system. Unfortunately, once a target configuration is selected, areas of the code that are not related to the selected software features will not be exercised by the build process. For example, since we focus on the Linux environment, Windows-specific code will be omitted during the build process. A static analysis of build specifications, such as that of Tamrawi et al. (2012) could be used to derive BDGs (and could easily be plugged into our approach), however appropriate edge weights need to be defined and calculated for them.

The header file hotspot heuristic assumes that header files that have a high rebuild cost only become build performance problems when they change frequently. Poor build performance in infrequently changing header files still poses a lingering threat to build performance. However, the approach allows practitioners to configure the hotspot quadrant thresholds to match their build performance requirements.

8.4 External validity

We focus our case study on four open source systems, which threatens the generalizability of our case study results. However, we studied a variety of systems with different sizes and domain to combat potential bias.

The build systems of the studied systems rely on `make` specifications, which may bias our case study results towards such technologies. However, our approach is agnostic of the underlying build system, operating on a build dependency graph, which can be extracted from any build system. Furthermore, our study focuses on header file hotspots, which are a property of C/C++ systems for which `make`-based build systems are the de facto standard (McIntosh et al. 2015).

The thresholds that we selected for the quadrant plots threaten the reliability of our case study results. Use of different thresholds will produce different quadrants, and thus, different header file hotspots. However, we believe that our selected elapsed time thresholds are representative, since the values were derived from the literature (Fischer et al. 2005; Humble and Farley 2010). Moreover, we use the median for the rate of change threshold—a metric that is resilient to outliers.

9 Related work

Developers rely on the maintainability, correctness, and speed of the build system. We discuss the related work with respect to these dimensions below.

9.1 Maintainability

Prior work shows that keeping the build system in sync with the source code generates substantial project maintenance overhead. Kumfert and Epperly (2002) and Hochstein and Jiao (2011) find that there is a "hidden overhead" associated with maintaining build systems. In our prior work, we show that build systems tend to co-evolve with source code from release to release (Adams et al. 2008; McIntosh et al. 2012). We have also shown that build system evolution imposes a non-trivial overhead on software development (McIntosh et al. 2011), e.g. up to 27 % of source code changes require accompanying build changes. In this work, we study build performance, which is another tangible form of build system overhead on software development.

9.2 Correctness

Neglecting build maintenance when it is necessary can generate lingering inconsistencies in the build process. Nadi et al. (2013) find that Linux kernel variability anomalies, i.e., inconsistencies between source code, configuration, and build specifications are rarely caused by trivial, typo-related issues, but more often caused by incomplete changes, e.g. changes to configuration files that are not properly reflected in the source code. Furthermore, these variability anomalies tend to linger for six Linux releases before getting fixed. Neitsch et al. (2012) find that abstractions and concepts tend to leak between source code and build system. Complementing their work, we find that architectural decay lowers the explanatory power that code properties provide to models built to explain hotspot likelihood.

Prior research has proposed several tools to assist developers in maintaining correctness in the build system. Adams et al. (2007) develop the MAKAO tool to visualize and reason about build dependencies. Tamrawi et al. (2012) propose a technique for visualizing and verifying build dependencies using symbolic dependency graphs. Al-Kofahi et al. (2012) extract the semantics of build specification changes using MkDiff. Nadi and Holt (2011, 2012) develop a technique for reporting anomalies between source code and build system as likely defects in Linux. We propose an automatable approach that establishes the basis for a tool capable of detecting header file hotspots.

When the build system fails, it causes build breakage, which can slow development progress. Hassan and Zhang (2006) use decision trees to predict whether a build will pass a lengthy (occasionally day-long) build certification process. Wolf et al. (2009) use social network analysis to show that team communication can predict broken builds. Kwan et al. (2011) show that socio-technical congruence, i.e., the agreement between technical dependencies and social alignment of a software team can also predict broken builds. Van der Storm (2008) uses a backtracking algorithm to prevent continuous integration builds from impacting developers. Similar to our rebuild cost metric, van der Storm (2007) defines a *build penalty* metric to measure the cost of interface changes in component-based software systems. While these studies focus on troublesome broken builds that cost developers time, our approach narrows build optimization focus to frequently changing header files that trigger slow rebuild processes.

9.3 Speed

Prior work also examines how build processes can be accelerated. Adams et al. (1994) achieve up to 80 % improvement in build performance through intelligent recompilation algorithms and elimination of unused environment symbols. Yu et al. improve both incremental and full build speed by automatically removing unnecessary dependencies between files (Yu et al. 2005) and redundant code from C header files (Yu et al. 2003). Dayani-Fard et al. (2005) semi-automatically propose architectural refactorings that will improve build performance. Our work complements the prior studies by narrowing optimization focus to the most costly header file hotspots that can be targeted individually, rather than all at once.

Recent work has also explored the applicability of refactoring techniques to build specifications themselves. For example, Vakilian et al. (2015) devise a technique to detect and refactor *underutilized targets*, i.e., build targets that are only partially used by the targets that depend upon them. Underutilized targets may trigger updates to those targets that depend upon them, even if those updates are not strictly necessary. By dividing underutilized targets into smaller, independent ones, Vakilian et al. (2015) are able to improve build performance. We believe that our approach is complementary to the approach of Vakilian et al. (2015) in two ways: (1) our technique includes the change frequency dimension, which could be used to reduce the scope of the target refactoring to those targets that really make a difference in day-to-day development, and (2) applying the notion of decomposing underutilized targets to hotspot files may be an interesting way to address build hotspots. Furthermore, while the approach of Vakilian et al. (2015) focuses on refactoring build specifications, our technique focuses on files, components, and subsystems that would benefit from build optimization effort.

10 Conclusions

Developers rely on the build system to produce testable deliverables in a timely fashion. A fast build system is at the heart of modern software development. However, software systems are large and complex, often being composed of thousands of source code files that must be carefully translated into deliverables in a timely fashion by the build system. As software projects age, their build systems tend to grow in size and complexity, making build profiling and performance analysis challenging.

In this paper, we propose an approach for pinpointing header file hotspots in C/C++ systems by analyzing both the build dependency graph and the change history of a software system. Our approach can be used to prioritize build optimization effort, allowing teams to focus effort on the header files that will deliver the most value in return. By continuously (re)applying our approach, development teams can verify that build optimization effort has indeed had an impact. Through a case study on four open source systems, we show that:

– The header file hotspot approach highlights header files that, if optimized, yield more improvement in the future total rebuild cost than just the header files that trigger the slowest rebuild processes, change the most frequently, or are used the most throughout the codebase (Sect. 6).
– Regression models are capable of explaining between 32 and 57 % of the detected hotspots using code layout and content properties of the header files (Sect. 7).
– In large projects, build optimization benefits more from architectural refinement than from acting on code properties like header file fan-in alone (Sect. 7).

References

Adams, B., De Schutter, K., Tromp, H., Meuter, W.: Design recovery and maintenance of build systems. In: Proceedings of the 23rd International Conference on Software Maintenance (ICSM), pp. 114–123 (2007)

Adams, B., Schutter, KD., Tromp, H., Meuter, WD.: The evolution of the linux build system. In: Electronic Communications of the ECEASST 8 (2008)

Adams, R., Tichy, W., Weinert, A.: The cost of selective recompilation and environment processing. Trans. Softw. Eng. Methodol. (TOSEM) **3**(1), 3–28 (1994)

Al-Kofahi, J.M., Nguyen, H.V., Nguyen, A.T., Nguyen, T.T., Nguyen, T.N.: Detecting semantic changes in makefile build code. In: Proceedings of the 28th International Conference on Software Maintenance (ICSM), pp. 150–159 (2012)

Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, work dependencies, and their impact on failures. Trans. Softw. Eng. (TSE) **35**(6), 864–878 (2009)

Chambers, J.M., Hastie, T.J. (eds.): Statistical Models in S, vol. 4. Wadsworth and Brooks/Cole, Pacific Grove (1992)

Dayani-Fard, H., Yu, Y., Mylopoulos, J., Andritsos, P.: Improving the build architecture of legacy C/C++ Software systems. In: Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 96–110 (2005)

Feldman, S.: Make: a program for maintaining computer programs. Software **9**(4), 255–265 (1979)

Fischer, A.R.H., Blommaert, F.J.J., Midden, C.J.H.: Monitoring and evaluation of time delay. Int. J. Hum. Comput. Interact. **19**(2), 163–180 (2005)

Fox, J.: Applied Regression Analysis and Generalized Linear Models, 2nd edn. Sage Publications, Thousand Oaks (2008)

Hassan, A.E., Zhang, K.: Using decision trees to predict the certification result of a build. In: Proceedings of the 21st International Conference on Automated Software Engineering (ASE), pp. 189–198 (2006)

Hochstein, L., Jiao, Y.: The cost of the build tax in scientific software. In: Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 384–387 (2011)

Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, New Jersey (2010)

Khomh, F., Chan, B., Zou, Y., Hassan, A.E.: An Entropy evaluation approach for triaging field crashes: a case study of mozilla firefox. In: Proceedings of the 18th Working Conference on Reverse Engineering (WCRE), pp. 261–270 (2011)

Kumfert, G., Epperly, T.: Software in the DOE: the hidden overhead of “The Build”. Techical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA (2002)

Kwan, I., Schröter, A., Damian, D.: Does socio-technical congruence have an effect on software build success? A study of coordination in a software project? Trans. Softw. Eng. (TSE) **37**(3), 307–324 (2011)

Lakos, J.: Large-Scale C++ Software Design. Addison-Wesley, New Jersey (1996)

McIntosh, S., Adams, B., Nguyen, T.H.D., Kamei, Y., Hassan, A.E.: An empirical study of build maintenance effort. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), pp. 141–150 (2011)

McIntosh, S., Adams, B., Hassan, A.E.: The evolution of Java build systems. Empir. Softw. Eng. **17**(4–5), 578–608 (2012)

McIntosh, S., Nagappan, M., Adams, B., Mockus, A., Hassan, A.E.: A large-scale empirical study of the relationship between build technology and build maintenance. Empir. Softw. Eng. (2015)

Mockus, A.: Organizational volatility and its effects on software defects. In: Proceedings of the 18th Symposium on the Foundations of Software Engineering (FSE), pp. 117–126 (2010)

Morgenthaler, J.D., Gridnev, M., Sauciuc, R., Bhansali, S.: Searching for build debt: experiences managing technical debt at google. In: Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD), pp. 1–6 (2012)

Nadi, S., Holt, R.: Make it or break it: mining anomalies in linux kbuild. In: Proceedings of the 18th Working Conference on Reverse Engineering (WCRE), pp. 315–324 (2011)

Nadi, S., Holt, R.: Mining Kbuild to detect variability anomalies in linux. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 107–116 (2012)

Nadi, S., Dietrich, C., Tartler, R., Holt, R.C., Lohmann, D.: Linux variability anomalies: what causes them and how do they get fixed? In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), pp. 111–120 (2013)

Neitsch, A., Wong, K., Godfrey, M.W.: Build system issues in multilanguage software. In: Proceedings of the 28th International Conference on Software Maintenance, pp. 140–149 (2012)

R Core Team (2013) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. http://www.R-project.org/

Shihab, E., Jiang, Z.M., Ibrahim, W.M., Adams, B., Hassan, A.E.: Understanding the Impact of code and process metrics on post-release defects: a case study on the eclipse project. In: Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–10 (2010)

van der Storm, T.: Component-based configuration, integration and delivery. Ph.D Thesis, University of Amsterdam (2007)

van der Storm, T.: Backtracking incremental continuous integration. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), pp. 233–242 (2008)

Tamrawi, A., Nguyen, H.A., Nguyen, H.V., Nguyen, T.: Build Code analysis with symbolic evaluation. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), pp. 650–660 (2012)

Tu, Q., Godfrey, M.W.: The build-time software architecture view. In: Proceedings of the 17th International Conference on Software Maintenance (ICSM), pp. 398–407 (2001)

Vakilian, M., Sauciuc, R., Morgenthaler, J.D., Mirrokni, V.: Automated decomposition of build targets. In: Proceedings of the 37th International Conference on Software Engineering (ICSE), pp. 123–133 (2015)

Wolf, T., Schröter, A., Damian, D., Nguyen, T.: Predicting build failures using social network analysis on developer communication. In: Procedings of the 31st International Conference on Software Engineering (ICSE), pp. 1–11. Washington, DC (2009)

Yu, Y., Dayani-Fard, H., Mylopoulos, J.: Removing false code dependencies to speedup software build processes. In: Proceedings of the 13th IBM Centre for Advanced Studies Conference (CASCON), pp. 343–352 (2003)

Yu, Y., Dayani-Fard, H., Mylopoulos, J., Andritsos, P.: Reducing build time through precompilations for evolving large software. In: Proceedings of the 21st International Conference on Software Maintenance (ICSM), pp. 59–68 (2005)