An Empirical Study of Build Maintenance Effort

Shane McIntosh, Bram Adams, Thanh H. D. Nguyen,
Yasutaka Kamei, and Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{mcintosh, bram, thanhnguyen, kamei, ahmed}@cs.queensu.ca

ABSTRACT

The build system of a software project is responsible for transforming source code and other development artifacts into executable programs and deliverables. Similar to source code, build system specifications require maintenance to cope with newly implemented features, changes to imported Application Program Interfaces (APIs), and source code restructuring. In this paper, we mine the version histories of one proprietary and nine open source projects of different sizes and domain to analyze the overhead that build maintenance imposes on developers. We split our analysis into two dimensions: (1) Build Coupling, i.e., how frequently source code changes require build changes, and (2) Build Ownership, i.e., the proportion of developers responsible for build maintenance. Our results indicate that, despite the difference in scale, the build system churn rate is comparable to that of the source code, and build changes induce more relative churn on the build system than source code changes induce on the source code. Furthermore, build maintenance yields up to a 27% overhead on source code development and a 44% overhead on test development. Up to 79% of source code developers and 89% of test code developers are significantly impacted by build maintenance, yet investment in build experts can reduce the proportion of impacted developers to 22% of source code developers and 24% of test code developers.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming environments—*Programmer workbench*; D.2.9 [**Software Engineering**]: Management—*Productivity, Programming teams, Software configuration management*

General Terms

Management, Measurement

Keywords

Empirical software engineering, build systems, mining software repositories

1. INTRODUCTION

The build system of a software project is the infrastructure that translates source code, libraries, and data files into a set of deliverables (e.g., executables and documentation) that are ready for distribution to customers. This transformation into deliverables may involve thousands of build commands that must be executed in a specific order to ensure the validity of the end product.

The build system is at the heart of the software development ecosystem. First, developers need to run the build system dozens of times per day to test the impact that their code changes have on the software product. Second, the build system is responsible for co-ordinating the execution of unit tests, for example to only run the most critical tests for a given source code change instead of the entire test suite. Third, the complex task of packaging a software product for release is typically automated by the build system, ensuring that the correct versions of software components, required libraries, documentation, and data files are included in the release. Fourth, the practice of continuous integration, i.e., regularly downloading the latest source code changes onto a dedicated server to validate that all unit and integration tests still pass, would be impossible without a robust build system that co-ordinates deliverable construction, test suite execution, and test report generation.

Despite the critical role of the build system, build maintenance effort and its overhead on developers is still largely undocumented. Of course, every seasoned programmer has build system horror stories, ranging from cryptic build error messages to subtle inconsistencies in build deliverables. Prior research on a number of small software projects estimates that build maintenance imposes a 12% overhead on the development process [13], distracting developers from their main tasks. Other researchers studied the build system of the Linux kernel, and found that the Linux build engineers have spent a considerable amount of time to make their build system as simple as possible for developers, at the expense of a very complex and hard to maintain core of build system machinery [2]. Recently, large software systems such as MySQL [9], Second Life [14], and KDE [22], have migrated from older build system technologies like `make` to newer build technologies like `cmake` to reduce the impact that tedious build maintenance has on the productivity of developers.

To provide project managers and developers with tangible measurements of the overhead of build maintenance, this paper performs a detailed empirical study of ten large, long-lived C and Java projects, including one proprietary Java system. We study the logical coupling [8] between source code and build system changes across individual revisions as well as across groups of revisions that resolve a *work item* (e.g., an enhancement or a bug fix). We find that: (1) the build system accounts for a relatively small proportion of the files in a project (9% median), yet has a comparable churn rate to the source code, suggesting that build systems are continually evolving and are likely to have defects [20], and (2) up to 27% of the work items involving production code changes (i.e., changes to system functionality), and 44% of the work items involving test code changes (i.e., changes to the automated tests and the test automation framework) require build maintenance.

The take-home message of this paper is that managers for C projects should explicitly account for up to 27% of the work items involving production code changes to require build maintenance. Managers for Java projects can expect that 4–16% of the work items involving production code changes to require build maintenance. The analyzed projects adopt one of two build ownership styles to cope with build maintenance: (1) most build changes are performed by a small team of build experts, or (2) build changes are dispersed amongst the development team.

The main contributions of this paper are empirical studies in ten large, long-lived software systems to analyze:

1. The size and churn rate of the build system.
2. The coupling between production and test code with the build system at the revision and work item levels.
3. Build ownership distribution across developers.

Paper Organization – Section 2 motivates our study of build maintenance, followed by a preliminary analysis of build maintenance in Section 3. Section 4 studies the logical coupling of production and test code changes with build changes, while Section 5 studies how projects distribute the build maintenance work. The threats to the validity of our work are presented in Section 6. Section 7 surveys related work. Finally, Section 8 presents the conclusions.

2. THE CRITICAL ROLE OF THE BUILD

The build system plays an important role in software development, since most stakeholders in the software development process interact with it [24].

First off, developers use a build system on a daily basis to test a software system after adding a new feature or fixing a bug. To minimize the time spent waiting while the build executes, build systems typically provide incremental build modes. Still, the interaction between build systems and developers is a major source of frustration. To illustrate this frustration, we analyzed the bug repositories of Mozilla, ArgoUML, and Eclipse. For example, Mozilla defect 351377 [17] describes a case where one Mozilla developer had removed an obsolete part of the build code and tested his changes locally without any issue. However, when another developer merged the build changes with his working copy, he could no longer run the build because of subtle differences between his build environment and the first developer's one. The second developer (and conceivably many others) could no longer build or test changes locally until this build issue was resolved.

A second important interaction of the build system is with the Continuous Integration (CI) server [26]. This server regularly downloads the latest source code changes to run all automated unit and integration tests, with the aim of identifying defects in new source code as soon as possible. To accomplish this, CI servers execute project builds that first construct the deliverables, then run the automated tests, and finally generate extensive test reports. Again, many problems can appear during this interaction with the build system. For example, build number 175 on the ArgoUML CI server failed to complete [23]. Developer time and effort was then invested to determine which change was responsible for the build failure, and what corrective action was needed. It took the developers two days to determine the root cause of the build failure. This investigation diverted developer attention away from the core tasks of fixing bugs and adding new features.

A third important interaction of the build system is with software release management [7]. The build system is responsible for bundling software deliverables into an installable package. All necessary components of the product should be bundled together, i.e., the correct versions of required libraries, data files, product documentation, and constructed deliverables. Since the released package is the final product of software development, any build error propagates directly to the customer. For example, in Firefox 3.0, issue 417037 [29] prevented users in a networked environment from accessing web pages via the address and search bars, the core feature of any web browser. Users in this networked environment could not use the Firefox 3.0 product until the release of the first service pack, i.e., 3.0.1, four months later. It turned out that a change to the Firefox build system caused executables to link against an incorrect version of the SQLite library, leading to inconsistencies in the Firefox 3.0 package.

Since build systems play a critical role in software development, build systems are typically hard to debug and understand [3, 30], and build system problems can have a major impact, even on the end user, it is important to understand how build systems are maintained in practice. Similar to Kumfert *et al.* [13], we assume that build maintenance is not accounted for in a project's budget (it is a hidden cost), hence we consider any development activity requiring build changes as overhead on development progress. Empirically measuring this overhead is the goal of this paper.

3. PRELIMINARY ANALYSIS

In prior work, we showed that build systems evolve in size and complexity, and that there is a correlation between the size and complexity of the source and build files [2, 16]. However, this prior analysis was performed on releases, not on revisions in code repositories, and we did not distinguish between test and production code. Hence, although we showed a correlation, it is not certain that individual developer actions in between releases, such as adding a file or restructuring a method, are directly coupled to build changes.

To address this gap, we perform a revision-level analysis focused on the following three Preliminary Questions (PQs):

PQ1) How many files does a typical build system consist of? We want to study the size of a typical build system to better understand the magnitude of build maintenance.

Table 1: Listing of the studied projects. Asterisks (*) denote previously used build technologies.

	ArgoUML	Hibernate	Eclipse	Jazz	GCC	Git	Linux	Mozilla	PLplot	PostgreSQL
Timespan	'98-'09	'01-'07	'01-'10	'07-'08	'88-'05	'05-'09	'05-'10	'98-'10	'92-'09	'96-'09
Program lang	Java	Java	Java	Java	C	C	C	C	C	C
Build techs.	Make* ANT	ANT* Maven	PDE Build	PDE Build	Autotools	Make Autoconf	Make KConfig	Make Autoconf	Make* Autotools* CMake	Autotools
# Build Files (BF)	614	211	483	5,967	1,719	43	3,726	10,709	652	771
# Prod Files	7,116	9,272	2,391	45,275	14,181	743	42,912	43,952	659	2,683
# Test Files	891	7,426	1,211	14,738	21,109	824	340	30,835	791	1,377
Total (TS)	8,007	16,698	3,602	60,013	35,290	1,567	43,252	74,787	1,450	4,060
$\frac{BF}{BF+TS}$	7%	1%	12%	10%	5%	3%	8%	12%	31%	16%

Table 2: File type classification examples.

Build	Production	Test
Build code (Makefile*, configure*)	Production code (*.c*, *.h*, *.java)	Unit tests (*.c*, *.h*, *.java)

PQ2) How much does a typical build system churn?
Churn measures the rate of change in source code. Prior studies have found that frequently changing source code, i.e., code with high churn, has a higher defect density [20] and causes more post-release defects [21]. We want to measure churn in the build system to gain insight into how susceptible the build system is to defects.

PQ3) How large are typical build system changes?
There is no prior work that quantifies the size of typical build system changes. Large changes imply that considerable effort is put into build maintenance.

3.1 Studied Projects

To combat potential bias in our results, we conduct a large scale study of projects of different sizes and domain, implemented with different programming languages and build technologies, prescribing to proprietary and open source processes. Table 1 gives an overview of the studied projects.

ArgoUML is a computer-aided software engineering tool for producing Unified Modelling Language (UML) diagrams. Hibernate is an object-to-relational mapping framework for Java programs, of which we examined the "core" subproject. Eclipse is an open source Integrated Development Environment (IDE), of which we studied the "core" subsystem. Jazz $^{\text{TM}1}$ is a proprietary next-generation IDE developed by IBM. The GNU Compiler Collection (GCC) is a popular source code compiler with front-ends for many programming languages. Git is a Distributed Version Control System (DVCS). Linux is an operating system kernel. Mozilla is a suite of internet tools, including (amongst others) the Firefox web browser. PLPlot is a plotting library with bindings for many popular programming languages. PostgreSQL is an object-relational database system.

3.2 Approach

Figure 1 shows an overview of our approach.

[1]`http://www.jazz.net`. IBM and Jazz are trademarks of IBM Corporation in the US, other countries, or both.

Data preprocessing – After downloading the VCS repositories of the ten studied projects, we extract the list of all revisions that have been committed by developers throughout the analyzed timespan, as well as the list of all files that ever existed. We wrote a dedicated shell script for this extraction.

File type classification – We classify each unique file that ever existed in the analyzed timespan as either build, production, or test code. Those files that do not fit in any category are marked as "other". Table 2 provides example classifications for some types of files. To facilitate future studies, we have put the resulting classifications online [1].

The classification process was semi-automated. Most files could be classified using file type naming conventions, e.g., `Makefile`s were marked as build files. However, patterns such as ".java" and ".xml" were ambiguous, since some .java files are production code while others are test code. After initial filtering of unambiguous file types, the remaining files had to be classified manually. For example, of the 49,364 files in Linux, approximately 40,000 could be classified automatically. The remaining 9,000 or so files had to be tagged manually based on our inspection of the files and prior experience with build systems [2, 3, 16].

The GCC, Git, Mozilla, PLplot, and PostgreSQL projects make use of the GNU Autotools and CMake build abstraction languages. These languages allow developers to implement build logic for many different platforms using an abstract representation of the build process. A build code generator produces the necessary platform-specific build code at build-time. For these case studies, we focus our analysis on revisions to the GNU Autotools and CMake abstraction files, not to the generated `Makefile`s and `configure` scripts, since the generated files are rarely recorded in the VCS. In the rare cases that generated files are (accidentally) recorded in the VCS, they introduce noise in our data. This noise turns out to be limited in practice.

Work item aggregation – By examining the messages of each revision, we were able to group revisions according to the Issue Tracking System (ITS) work item ID that they collectively resolve. Work item aggregation is performed to provide a coarser granularity of analysis representing a single unit of development work instead of single revisions.

Item/Revision classification – Equipped with the list of classified files, the VCS revisions and work items can be classified. Unlike the file type classification, a revision or work item can fit into many categories. For example, if a re-

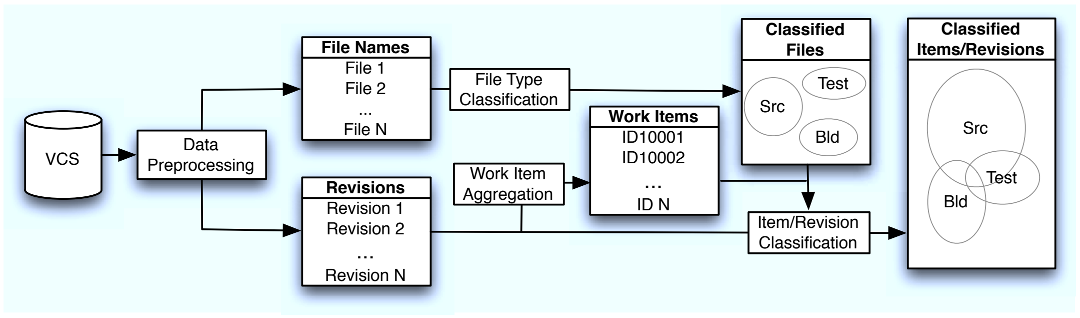

Figure 1: Overview of our approach to study build maintenance effort.

vision or work item modifies both build and production files, it is marked as both a build and production code modifying revision or work item.

PQ1 – The build system accounts for 9% of the maintained files

Table 1 shows that the build system accounts for 1-31% of all maintained files that ever existed in the analyzed timespan, with a median of 9%. These low values indicate that in most cases (PLplot being the exception), the build system is dwarfed by the other development artifacts.

Hibernate-core (1%) and PLplot (31%) are anomalies. Being entirely composed of a single library, the Hibernate-core project has little build code (211 files), which explains the low 1%. On the other hand, the PLplot project has the most inflated build file percentage (31%). While the PLplot project is rather small, it provides bindings to many programming languages. Each binding has its own build component, increasing the build system size. The problem is compounded by two build technology migrations that PLPlot has undergone. The migrations reimplemented build code from `make` into GNU Autotools, and later from GNU Autotools into CMake.

PQ2 – Normalized churn rates of build and source code files are comparable

To study the churn rate in the build system, we compare it against that of the source code. To account for the discrepancy in size between source and build files (PQ1), we measure the churn rate in the source code and build system using *normalized churn*. First, we count the number of source files and the number of build files that were changed in each month-long development period. Then, we divide each count by the total number of source files or the total number of build files that existed in that period. We repeat this process for each month. We chose a month-long period rather than shorter periods, such as a day or a week, because the latter two are typically too short for a significant amount of development to occur.

Figure 2 plots the distribution of the monthly normalized churn using a *beanplot*. Beanplots are boxplots in which the vertical curves summarize and compare the distributions of different data sets [10], which in our case correspond to the normalized churn of build and source code in month-long development periods. The horizontal black lines indicate the median of the normalized change for each project's source (black) and build files (grey).

In most of the studied projects, the median of the monthly normalized change for the source and build files are relatively close to each other, only differing by at most 7% (GCC). Hibernate-core is the only project with a median value of the normalized churn for the build files greater than that of the source files. The Hibernate-core project had only 1-7 build files during the first 12 months of development, and easily reached 100% normalized churn, skewing the median.

The comparable churn in the source and build files is concerning, since prior research on source code has shown that frequently churning modules contain more defects than slowly changing ones [20]. Build maintainers must be very diligent to ensure that the build system does not become defect-prone, since a defect-ridden build system may: (1) slow development progress due to suboptimal build routines [18], and (2) fail to produce correct deliverables, grinding development progress to a halt [11, 23].

PQ3 – Build changes induce more relative churn than source code changes do

Table 3 shows the median of the number of Lines added and deleted Per Revision (LPR) in nine of the studied projects. Jazz data was unavailable for analysis because we do not have access to the actual source code. We also normalize the median LPR by the number of build and source files respectively, yielding the Lines added and deleted per Revision and File (LPRF). Since the LPRF values are quite small, we report them as a percentage, i.e., multiply by 100.

Most source code changes add 8–17 lines and delete 4–9 lines (median). The corresponding numbers for build changes are 2–6 lines and 1–5 lines. Thus, the size of a typical build change is about 1/4–1/2 of the size of a typical source change. However, the LPRF values show that build changes induce more relative churn on the build system than source code changes induce on the source code. PLplot presents the only exception, since the LPRF for added lines is smaller for the build system than for the source code. This is due to the inflated size of the PLplot build system, since it endured two build technology migrations (see PQ1).

3.3 Discussion

We find that the build system is relatively small in size, being composed of 9% (median) of the files in the studied projects (PQ1). However, it churns frequently, at rates similar to the source code, suggesting that an overhead is imposed upon developers (PQ2). Build maintenance changes typically add 2–6 and delete 1–5 lines, and generate more relative churn for the build system than the source code changes generate for the source code (PQ3).

To analyze whether these revision-level build changes are burdensome on developers, we now investigate how tightly coupled developer changes to production and test code are to build changes. We focus our study of build maintenance

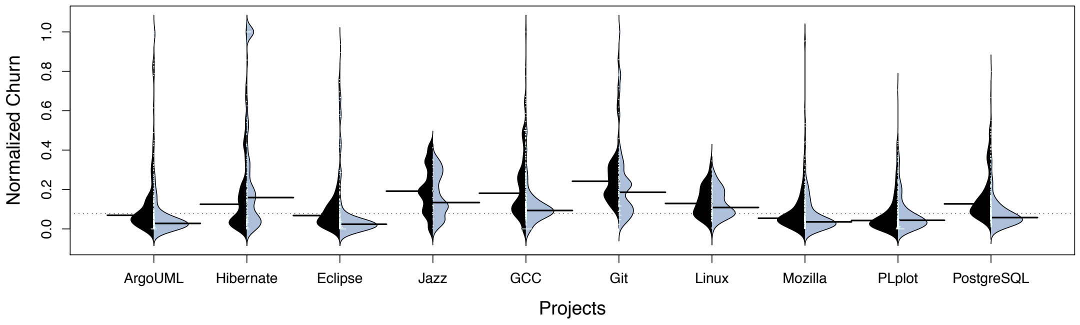

Figure 2: Distribution of monthly churn in source (black) and build (grey) files.

Table 3: Median LPR (#) and LPRF (%) for all projects except Jazz.

		ArgoUML		Hibernate		Eclipse		GCC		Git		Linux		Mozilla		PLplot		PostgreSQL	
		+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
Bld	LPR	2	2	4	2	3	2	6	5	2	1	2	2	4	3	5	4	4	3
	LPRF	0.33	0.33	1.90	0.95	0.62	0.41	0.35	0.29	4.65	2.32	0.05	0.05	0.04	0.03	0.77	0.61	0.52	0.39
Prod	LPR	15	9	17	9	10	5	8	5	8	4	9	6	11	6	14	7	15	9
	LPRF	0.19	0.11	0.10	0.05	0.28	0.14	0.02	0.01	0.51	0.26	0.02	0.01	0.01	0.01	0.97	0.48	0.37	0.22

overhead on the following two Research Questions (RQs):

RQ1) How tight is the coupling between source code and build system? (Section 4)
Kumfert *et al.* estimate that developers spend 12% of their time keeping the build system in sync with the source code, rather than fixing bugs and adding new features [13]. These results are mainly based on a survey asking developers about their overall build maintenance investment. We are interested in rigorously validating these findings with the actual changes developers make.

RQ2) How is build ownership distributed amongst developers? (Section 5)
Since build files churn frequently, projects such as Linux [2] and Perl [30] designate members of the development team as build experts. To study the different ways in which projects allocate personnel to build maintenance, we want to see how many developers have to change the build.

4. BUILD COUPLING (RQ1)

In this section, we measure how tight the coupling is between the build files and production or test files.

4.1 Approach

We adopt association rules to measure the relationships between production, test, and build files. An association rule is a statistical description of co-occurring elements in a dataset [4]. For example, Amazon.com recommends additional purchases to a customer by mining association rules from their database of prior customer purchases. In our case, we do not mine new association rules, but rather we evaluate the following specific associations:

Prod $\Rightarrow$ **Bld** measures the implication that a production code change will be accompanied by a build change in the same revision or work item. Similarly, **Test** $\Rightarrow$ **Bld** measures the implication that a test code change will be accompanied by a build change in the same revision or work item.

Bld $\Rightarrow$ **Prod** measures the implication that a build change will be accompanied by a production code change in the same revision or work item. Intuitively, we expect this implication to be strong, since our prior work [16] showed that the majority of build maintenance is the result of production code changes. Hence, we expect that build changes should be grouped with the source code changes that initiated it. Similarly, **Bld** $\Rightarrow$ **Test** measures the implication that a build change will require an accompanying test code change.

Table 4: Association rule interest metrics.

Metric	Calculation
Support(A)	$\frac{\#\ class\ A\ revisions}{\#\ total\ revisions}$
Conf(A $\Rightarrow$ B)	$\frac{Support(A,B)}{Support(A)}$
Conv(A $\Rightarrow$ B)	$\frac{Support(A) \times Support(\neg B)}{Support(A,\neg B)}$

We evaluate the association rules above using the "interest" metrics in Table 4. Support(X) is defined as the proportion of revisions or work items that contain X [4]. The coupling or confidence metric Conf(X $\Rightarrow$ Y) measures the strength of the implication that a change to Y will occur within the same revision or work item as a change to X [4]. Gall *et al.* use an identical metric to measure logical coupling between source code files [8]. Note that confidence measures are asymmetrical, i.e., Conf(X $\Rightarrow$ Y) $\neq$ Conf(Y $\Rightarrow$ X).

We use a χ^2 goodness-of-fit test [27] to validate the statistical significance of the coupling between changes to X and Y. If the χ^2 statistic is greater than 3.84 ($\alpha = 0.05$), the relationship is statistically significant. Otherwise, the observed relationship is due to chance. We report the p-value of the χ^2 test, rather than the χ^2 statistic. The χ^2 statistic is symmetrical.

Table 5: Revision-level interest metric values for production, test, and build code.

		ArgoUML	Hibernate	Eclipse	Jazz	GCC	Git	Linux	Mozilla	PLplot	PostgreSQL
Support	Prod	0.62	0.62	0.68	0.69	0.56	0.61	0.87	0.70	0.39	0.55
	Test	0.06	0.32	0.23	0.18	0.13	0.11	0.01	0.08	0.19	0.10
	Bld	0.07	0.08	0.08	0.09	0.15	0.07	0.10	0.16	0.36	0.16
	Prod, Bld	0.01	0.03	0.02	0.03	0.04	0.03	0.06	0.06	0.03	0.05
	Test, Bld	<0.01	0.02	0.01	0.01	0.01	<0.01	<0.01	0.01	0.03	0.02
Conf	Prod ⇒ Bld	0.02	0.05	0.03	0.04	0.07	0.04	0.06	0.08	0.08	0.10
	Bld ⇒ Prod	0.16	0.36	0.28	0.28	0.27	0.41	0.56	0.35	0.09	0.34
	Test ⇒ Bld	0.05	0.05	0.03	0.07	0.07	0.04	0.13	0.16	0.17	0.19
	Bld ⇒ Test	0.04	0.20	0.09	0.13	0.06	0.07	0.01	0.08	0.09	0.11
Conv	Prod ⇒ Bld	0.95	0.96	0.95	0.94	0.92	0.98	0.96	0.91	0.69	0.93
	Bld ⇒ Prod	0.45	0.59	0.44	0.43	0.60	0.66	0.30	0.46	0.67	0.68
	Test ⇒ Bld	0.98	0.97	0.95	0.97	0.91	0.97	1.04	1.00	0.77	1.03
	Bld ⇒ Test	1.02	0.91	1.34	0.94	0.96	0.96	1.01	1.02	0.97	1.05
χ^2 (p-value)	Prod, Bld	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	Test, Bld	0.06	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.93	<0.01	<0.01

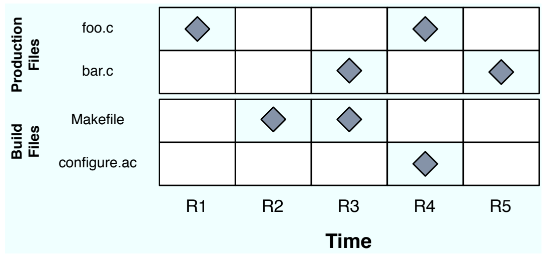

Figure 3: An example scenario.

Finally, the conviction metric Conv(X ⇒ Y) measures the departure of Conf(X ⇒ Y) from independence [6]. Conviction values less than one indicate that X is negatively correlated with Y, i.e., Conf(X ⇒ ¬ Y) > Conf(X ⇒ Y) [12]. Conviction values greater than one indicate that X is positively correlated with Y, i.e., Conf(X ⇒ Y) > Conf(X ⇒ ¬ Y). Throughout this study, we use conviction to evaluate whether the statistically significant logical coupling relationships between X and Y (indicated by χ^2) are positively or negatively correlated.

We use the example in Figure 3 to illustrate the confidence and conviction metrics. A series of five revisions appear on the Time axis and a series of four files (two production and two build files) appear on the Y-axis. Of the four production file revisions, two have build file changes as well (R3 and R4), thus the Conf(Prod ⇒ Bld) is 0.5. The Conf(Bld ⇒ Prod) is 0.67, since two of the three build revisions have a production file change. We cannot calculate the χ^2 value for this example, since the test was intended for a larger sample size. However, suppose that the p-value for the production-build relationship is smaller than 0.05. This would indicate that the low coupling and conviction of Prod ⇒ Bld would be statistically significant, i.e., not just an artifact of noise. The Conv(Prod ⇒ Bld) of 0.8 (based on a more complex calculation) indicates that Prod ⇒ ¬ Bld is a stronger rule than Prod ⇒ Bld, i.e., production code changes are typically not accompanied by build changes.

4.2 Revision-level Results

In this section, we measure the revision-level coupling between build and production or test files.

Low revision-level coupling from the production or test code to the build system: The χ^2 statistic in Table 5 shows that the coupling values from production to build files are statistically significant. However, the confidence values in Table 5 indicate that the coupling from production to build files ranges between 2-10%, i.e., no project reaches Kumfert *et al.*'s estimate of 12% [13]. The median value of our study is 5.5%, less than half of the 12% estimate.

The reason for the low coupling is that a revision rarely includes both production and build file changes, as indicated by the low Support(Prod, Bld) values. The low Conv(Prod ⇒ Bld) values indicate that Prod ⇒ ¬ Bld is a stronger rule than Prod ⇒ Bld. Especially in PLplot, the Conv(Prod ⇒ Bld) and Conv(Test ⇒ Bld) values indicate that production and test revisions are more likely not to include build file changes than vice versa. This is because of the two migration efforts that generated many build-only changes in PLplot.

There is low revision-level coupling from production and test code to the build system.

Low revision-level coupling from the build system to the production or test code: The statistically significant confidence values in Table 5 indicate that the build system is coupled more to the source code than vice versa. Yet, the Conv(Bld ⇒ Prod) values are all much less than 1 (0.43–0.68), indicating that it is still not likely that build revisions contain changes to the source code. This finding is counter-intuitive, since we would expect that most build changes would be accompanied by production code changes [16].

In PLplot, the confidence values are low due to one slow migration period, where a new build system was implemented while the existing build system was slowly phased out. Many of the build migration changes were committed in revisions that were not related to any source code, which affects the confidence values.

While the coupling from the build system to the production and test code is higher than in the other direction, it is still rather low.

4.3 Work Item Results

The confidence values that we have observed at the revision level suggest that there is less coupling between the production or test files and the build system than we had anticipated based on Kumfert *et al.*'s estimation [13]. However, our results agree with an earlier study of KDE that found that build revisions are often dominated by the build and do not co-change with other entities [28].

We conjecture that the low observed coupling is due to de-

Table 6: Overview of work item data.

	Eclipse	Jazz	Mozilla
Revisions	6,391	36,557	210,400
Revs with Work Items	4,092	22,485	79,242
% Revs with Work Items	64%	62%	38%
Work Items	2,452	11,611	55,199

Table 7: Work item interest metrics.

		Eclipse	Jazz	Mozilla
Support	Prod	0.87	0.85	0.83
	Test	0.31	0.24	0.17
	Bld	0.17	0.05	0.26
	Prod, Bld	0.14	0.04	0.22
	Test, Bld	0.06	0.02	0.08
Conf	Prod ⇒ Bld	0.16	0.04	0.27
	Bld ⇒ Prod	0.82	0.72	0.86
	Test ⇒ Bld	0.20	0.08	0.44
	Bld ⇒ Test	0.36	0.36	0.29
Conv	Prod ⇒ Bld	0.99	0.99	1.01
	Bld ⇒ Prod	0.74	0.52	1.15
	Test ⇒ Bld	1.03	1.03	1.31
	Bld ⇒ Test	1.07	1.19	1.16
χ^2 (p-value)	Prod, Bld	0.02	<0.01	<0.01
	Test, Bld	0.16	<0.01	<0.01

veloper commit behaviour. For example, while some developers commit related build and source code changes under one revision, others may commit build changes in separate revisions from source code changes, which introduces noise in the revision-level data. To address this, we should group all revisions that collectively resolve an ITS work item. By investigating the relationship between source and build files at the work item level, we aim to reduce the noise caused by inconsistent developer commit behaviour.

Our work item analysis is limited to the three projects in Table 6. The lack of availability of high quality work item linkage in VCS data is a known problem [5, 25]. With this in mind, a three project study is actually quite unique. Furthermore, Table 6 shows that a large portion of the VCS revisions of Eclipse-core (64%), Jazz (62%), and Mozilla (38%) could be linked to ITS work item IDs.

Production code work items are more tightly coupled to the build system in Mozilla than in the Eclipse-based projects: The Conf(Prod ⇒ Bld) and Conf(Test ⇒ Bld) values in Table 7 show that there is considerable coupling from the production and test code to the build system in Mozilla (27% and 44%). These numbers indicate that production code and build system consistency requires considerable developer participation, i.e., roughly one in every four work items requires build maintenance.

However, in Eclipse-core, the coupling is reduced to 16% and in Jazz the observed coupling is a mere 4%. Eclipse-core and Jazz both leverage the automated Eclipse Plugin Development Environment (PDE) build technology. Each Eclipse subsystem contains a "build.properties" file, which lists the high-level build system configuration. The PDE build parses these property files to either: (1) generate ANT scripts to perform the build appropriately, or (2) use an appropriate Eclipse plugin to perform the compilation and packaging. Since the developer must only maintain the high-level build.properties file (via the IDE), the daily build maintenance overhead is reduced.

Table 8: Build ownership interest metrics.

		Jazz	Git	Linux
	All	156	795	6,502
Support	Prod	0.81	0.85	0.97
	Test	0.36	0.22	0.02
	Bld	0.73	0.22	0.26
	Prod, Bld	0.63	0.19	0.24
	Test, Bld	0.32	0.05	0.01
Conf	Prod ⇒ Bld	0.79	0.22	0.25
	Bld ⇒ Prod	0.87	0.85	0.93
	Test ⇒ Bld	0.89	0.24	0.58
	Bld ⇒ Test	0.44	0.23	0.06
Conv	Prod ⇒ Bld	1.26	1.00	0.99
	Bld ⇒ Prod	1.46	0.98	0.48
	Test ⇒ Bld	2.51	1.02	1.76
	Bld ⇒ Test	1.14	1.02	1.03
χ^2 (p-value)	Prod, Bld	0.02	1.00	<0.01
	Test, Bld	0.01	0.95	<0.01

The Mozilla Conv(Test ⇒ Bld) value indicates that the logical coupling between test and build code of 44% is very strong, while the Conv(Test ⇒ Bld) values for Eclipse-core and Jazz are weaker. Note that the test and build code p-value for Eclipse-core is not statistically significant ($\alpha = 0.5$).

We find that there is a substantial coupling between production and test code with the build system in the Mozilla project. We observe a 19% increase in Conf(Prod ⇒ Bld) and a 28% increase in Conf(Test ⇒ Bld) over the revision-level analysis. We observe similar increases in Eclipse-core of 13% and 17% respectively. There was little change in the observed coupling for Jazz. We conjecture that the differences between Mozilla and Eclipse-core/Jazz are due to programming and project management differences, but more elaborate studies are needed to validate this conjecture.

The maintenance of the build system impacts both production and test development in Mozilla. The Eclipse and Jazz build code is automatically generated and edited via the IDE, resulting in reduced build system maintenance and coupling to the source code.

5. BUILD OWNERSHIP (RQ2)

Our study of Build Coupling reveals that Mozilla developers will have to perform build changes for roughly one in every four work items they are tasked with. However, we did not consider build ownership, i.e., which developers actually make those changes to the build system. Prior work reports that projects like the Perl interpreter [30] and the Linux kernel [2] dedicate a team of experts to build maintenance tasks. In those cases, although build coupling seems high, the work is delegated to build experts, such that developers will rarely need to perform the build maintenance.

We study the relationship between production, test, and build developers by evaluating the associations between them

Table 9: Number and percentage of developers responsible for 80% of the file changes to production, test, and build files.

	Jazz	Git	Linux
Prod	41 (26%)	57 (7%)	523 (8%)
Test	58 (37%)	95 (12%)	484 (7%)
Build	53 (34%)	44 (5%)	365 (5%)

with the Support, Confidence, χ^2, and Conviction "interest" metrics introduced for RQ1.

5.1 Approach

We first need to label authors as build, test, or source code developers. An author may hold one or more labels. We assume that developers who produce at least one source code revision are source code developers, since source code development is the main focus of a development team. However, we only label authors as build developers if their personal source-build coupling is greater than or equal to the project's source-build coupling. Similarly, we only label authors as test developers if their personal source-test coupling is greater than or equal to the project's source-test coupling. We choose such a definition to identify those developers responsible for a significant portion of build system (and test) development, excluding accidental contributors.

Our study is limited to projects that record correct author names. A common practice in open source development is to restrict VCS write access to only the core developers [5]. Many authors send patches, i.e., files containing their changes, to the core developers for review. After engaging in a review process, the core developer will write the changes to the VCS. Only the Git, Linux, and Jazz projects retain the name of the original author of the patch (instead of the core developer), so our analysis is limited to these three projects.

5.2 Results

We use Table 8 to illustrate two build ownership styles that projects adopt for maintaining the build system.

Concentrated and dispersed build ownership: We observe two styles of build ownership: (1) a concentrated style, where most of the build maintenance is performed by a small team of build engineers, and (2) a dispersed style where most developers contribute code to the build system. Style (1) was observed in the Linux and Git projects, while style (2) was observed in Jazz.

The Conf(Prod $\Rightarrow$ Bld) values of Git (22%) and Linux (25%) in Table 8 show that the majority of source code contributors in these two projects do not have to change the build system frequently, although the χ^2 statistic indicates that the Git coupling is not statistically significant. In Jazz, the build system rarely changes, but the few changes that happen are made by most production and test developers. Jazz's Support(Bld) value in Table 7 indeed shows that only 5% of all work items require build changes, however, the Conf(Prod $\Rightarrow$ Bld) values in Table 8 show that 79% of production code developers make a considerable number of changes to the build. In other words, although 79% of production code developers need to change the build, they do not need to do this often, i.e., the overhead of build maintenance on developers is limited.

Table 9 shows that to make 80% of all build changes, a smaller proportion of developers are needed in Git (5%) and Linux (5%) than in Jazz (34%). This indicates that build expertise is concentrated in the Git and Linux projects, whereas it is dispersed among developers in the Jazz project. Comparing the numbers to those of the production and test code, we see that the build consistently has the lowest proportion of developers that contribute 80% of the changes in the two open source projects (5%).

Since most of the build changes in Linux and Git are made by a core team of build experts, contributors are saved the hassle of build maintenance. In 2001, the Linux project in particular invested time and effort into reducing the impact that the build system had on contributors [2]. Our findings suggest that they were successful in concentrating build maintenance onto a core team of build experts.

While we do not have the data to speculate about which style performs best universally, one could conjecture that build ownership style (1) is more suitable for open source teams. Open source development depends on casual developer contributions. Casual developers will have a hard enough time learning the intricacies of foreign source code without having to struggle with the build system. Thus, offloading build maintenance onto a team of core engineers seems advisable. However, with limited resources, a build maintenance team can become a bottleneck for developers.

The studied projects adopt either a concentrated (Linux and Git), or dispersed (Jazz) build ownership style to cope with the build maintenance overhead.

Most test developers have to make build changes: The Conf(Test $\Rightarrow$ Bld) values in Table 8 reveal that 89% of Jazz test developers, 58% of Linux test developers, and 24% of Git test developers make a considerable number of changes to the build code. This indicates that the build system maintenance is impacting most of the test developers in Jazz and Linux. The corresponding χ^2 and conviction values for Jazz and Linux show that these percentages are statistically significant and the rule is positively correlated. Project managers should keep this in mind when performing test development planning and budget estimations.

6. THREATS TO VALIDITY

Construct validity: Our mapping of revisions to work items is based on project-specific heuristics. In addition, developers may have omitted or mistyped the work item IDs in revision comments [5, 25].

Internal Validity: We conjecture that by measuring the interestingness of the associations between production, test, and build files, we have measured the degree to which these entities co-evolve. It is possible that the phenomena that we observe are purely coincidental. However, the reported χ^2 and conviction values at the work item level suggest that these relationships are more than coincidental.

We assume that developers commit all related changes under the same revision or work item. We observed cases where developers mistakenly committed an incomplete revision and followed up later with a correction in a new revision. Such missteps introduce noise in our revision data. To combat this, we analyze the data at the work item level.

External Validity: We have only studied a limited number of projects and as such, our results may not generalize to all projects. However, we studied a wide variety of projects of different sizes and domain, prescribing to both proprietary and open source processes.

Unfortunately, no Microsoft .NET systems were available for study. Many .NET applications use MSBuild [19], a build technology where targets and tasks are described using XML (similar to ANT [16]). Hence, we are inclined to believe that MSBuild and ANT scripts would evolve similarly, but have no data to support this. In future work, we plan to validate this inclination empirically.

Reliability Validity: The replicability of our case studies is threatened by the subjectivity of our file classification approach. Some files that were not automatically classified based on file extension had to be classified manually, and as such were subject to the authors' opinion. To classify files that may have fit in more than one category, the authors' best judgement was used based on our prior experience with build systems [2, 3, 16]. To facilitate future studies, we have put the resulting classifications online [1].

7. RELATED WORK

In this section, we survey the related work in the fields of build system maintenance and logical coupling.

Kumfert *et al.* argue that the need to keep the build system in sync with the source code generates an implicit development overhead [13]. In Kumfert *et al.*'s survey, developers claim that 0-35.71% of their development time is spent maintaining the build system. Kumfert *et al.* validate one survey entry reporting a 20% overhead of the build by classifying each file as build- or source-related, and counting the number of revisions modifying each. We adopt a similar file classification and revision categorization approach, but use association rules and study to ten projects of varying size, implementation language, and domain. We also perform a work item analysis and study build ownership styles.

Robles *et al.* argue that software artifacts other than source code are often disregarded in software engineering research [28]. In a case study of KDE, they found that in revisions that contain build changes, build files are often the predominant file type, i.e., 50% or more of the files in the revision are build files, and many build revisions do not contain source code changes at all. Build files in KDE are tightly coupled to each other and weakly coupled to the source files. Similarly, in our study we found that there is low revision-level coupling from production and test code to the build system. However, when we lifted our analysis up to the work item level, we observed considerable coupling from production to build files and from test to build files.

In prior work, we studied the evolution of ANT build systems for four open source Java projects [16]. We found that both the complexity of the ANT scripts and the dynamic behaviour of the build system evolve. Furthermore, the build complexity co-evolves with the source code at the release level, complementing Adams *et al.*'s findings in the Linux kernel [2]. This paper measures the build maintenance overhead on the actual software development process.

Gall *et al.* study logical coupling between modules in product releases by measuring co-change between software modules [8]. We apply a similar technique, but rather than studying the coupling between modules, we study the logical coupling between production, test, and build files at the revision and work item level.

Zaidman *et al.* analyze the change history, growth history, and test quality evolution views of two open source systems to study the co-evolution of production and test code [31]. Our paper quantitatively studies the co-evolution of production code with the build system as well as the coupling of test code with the build system.

Lubsen *et al.* quantitatively study the co-evolution of production and test files by mining association rules from VCS revisions [15]. They group these rules into production and test code co-change classes and study the distribution of the rules. We adopt a similar approach, but rather than mining for low-level association rules, we extract support for specific association rules between production, test, and build files. We also find that work items (rather than revisions) are a more suitable level of granularity for identifying co-changing software entities.

8. CONCLUSIONS

This paper analyzes the coupling between the source code and build system changes in ten software projects, to measure the overhead of build maintenance on developers.

Our build coupling analysis shows that 4–16% of source code work items in the analyzed Java projects and 27% of source code work items in the analyzed C project require an accompanying build change. Furthermore, 8–20% of test code work items in the analyzed Java projects and 44% of test code work items in the analyzed C project require an accompanying build change. Project managers should explicitly account for build maintenance of this magnitude in their project plans and budgets. Furthermore, future co-evolution studies should consider analysis at the work item level, as we feel it more accurately represents the unit of development work.

Our analysis also shows that the normalized churn rates of the build system and source code are comparable. Moreover, build changes induce more relative churn on the build system than source code changes induce on the source code, which means that the build system may be susceptible to defects [20, 21].

We observe that the analyzed projects have two build ownership styles for coping with churn in the build system: (1) a small team of build experts handle most of the maintenance, or (2) maintenance is dispersed amongst most developers. In the three studied projects, the actual overhead of build maintenance for individual developers is limited. In future work, we plan to investigate the advantages and disadvantages of these two (and other) build ownership styles.

The analysis presented in this paper is primarily quantitative. However, our analysis framework can be used by future build system studies to: (1) compare the maintenance overhead of build tools and guide build tool improvements, and (2) identify which build ownership style performs best.

REFERENCES

[1] Classified files. `http://sailhome.cs.queensu.ca/~shane/Shane_McIntosh/Publications_files/classified_files.tar.bz2`.

[2] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. The evolution of the linux build system. *Electronic Communications of the ECEASST*, 8, 2008.

[3] B. Adams, K. De Schutter, H. Tromp, and W. Meuter. Design Recovery and Maintenance of Build Systems. In

Proc. of the 23rd Int'l Conf. on Software Maintenance (ICSM), pages 114–123, 2007.

[4] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *ACM SIGMOD Records*, 22(2):207–216, 1993.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proc. of the 7th European Software Eng. Conf/Sym. on the Foundations of Software Eng. (ESEC/FSE)*, pages 121–130, 2009.

[6] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In *Proc. of the 1997 ACM SIGMOD Int'l Conf. on Management Of Data*, pages 255–264. ACM, 1997.

[7] E. Dolstra, E. Visser, and M. de Jonge. Imposing a Memory Management Discipline on Software Deployment. In *Proc. of the 26th Int'l Conf. on Software Engineering (ICSE)*, pages 583–592. IEEE Computer Society, 2004.

[8] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proc. of the 14th Int'l Conf. on Software Maintenance (ICSM)*, pages 190–198. IEEE Computer Society, 1998.

[9] L. Grimmer. Building MySQL Server with CMake on Linux/Unix. `http://www.lenzg.net/archives/291-Building-MySQL-Server-with-CMake-on-LinuxUnix.html`, 2010. Last viewed: 08-Feb-2011.

[10] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 2008.

[11] KDE developer: "mosfet". Autoconf/Automake errors in kdelibs. `http://lists.kde.org/?l=kde-core-devel&m=95953244511288&w=4`. Last viewed: 08-Feb-2011.

[12] K. S. Krishna, P. R. Krishna, and S. K. De. Discovering fuzzy association rules with interest and conviction measures. In R. Khosla, R. J. Howlett, and L. C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3684 of *Lecture Notes in Computer Science*, pages 101–107. 2005.

[13] G. Kumfert and T. Epperly. Software in the DOE: The Hidden Overhead of "The Build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA, February 2002.

[14] Linden Labs. CMake. `http://wiki.secondlife.com/wiki/CMake`. Last viewed: 08-Feb-2011.

[15] Z. Lubsen, A. Zaidman, and M. Pinzger. Using Association Rules to Study the Co-evolution of Production & Test Code. In *Proc. of the 6th Working Conf. on Mining Software Repositories (MSR)*, pages 151–154. IEEE Computer Society, 2009.

[16] S. McIntosh, B. Adams, and A. E. Hassan. The Evolution of ANT Build Systems. In *Proc. of the 7th working conf. on Mining Software Repositories (MSR)*, pages 42–51. IEEE Computer Society, 2010.

[17] A. Miller. js/Makefile.in gone but still in allmakefiles.sh. `https://bugzilla.mozilla.org/show_bug.cgi?id=351377`. Last viewed: 08-Feb-2011.

[18] P. Miller. Recursive Make Considered Harmful. In *Australian Unix User Group Newsletter*, volume 19, pages 14–25, 1998.

[19] MSDN. The Microsoft Build Engine (MSBuild). `http://msdn.microsoft.com/en-us/library/dd393574.aspx`. Last viewed: 06-Feb-2011.

[20] N. Nagappan and T. Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proc. of the 27th Int'l Conf. on Software Engineering (ICSE)*, pages 284–292. ACM, 2005.

[21] N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Proc. of the 1st Int'l Symp. on Empirical Software Engineering and Measurement (ESEM)*, pages 364–373. IEEE Computer Society, 2007.

[22] A. Neundorf. Why the KDE project switched to CMake – and how (continued). `http://lwn.net/Articles/188693/`. Last viewed: 08-Feb-2011.

[23] T. Neustupny. Build failed in Hudson, what to do? `http://argouml.tigris.org/ds/viewMessage.do?dsForumId=450&dsMessageId=2618367`. Last viewed: 08-Feb-2011.

[24] G. V. Neville-Neal. Kode Vicious: System Changes and Side Effects. *Communications of the ACM*, 52(4):25–26, April 2009.

[25] T. H. D. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In *Proc. of the 17th Working Conf. on Reverse Engineering (WCRE)*, pages 259–268, 2010.

[26] G. Niemeyer and J. Poteet. *Extreme Programming with Ant: Building and Deploying Java Applications with JSP, EJB, XSLT, XDoclet, and JUnit*. Sams, 1st edition, 2003.

[27] J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury press, 1995.

[28] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo. Beyond Source Code: The Importance of Other Artifacts in Software Development (A Case Study). *Journal of Systems and Software (JSS)*, 79(9):1233–1248, 2006.

[29] T. Steiner. mozStorage chokes on databases over AFP. `https://bugzilla.mozilla.org/show_bug.cgi?id=417037`. Last viewed: 08-Feb-2011.

[30] Q. Tu and M. Godfrey. The build-time software architecture view. In *Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM)*, pages 398–407. IEEE Computer Society, 2002.

[31] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen. Mining Software Repositories to Study Co-Evolution of Production & Test Code. In *Proc. of the 1st Int'l Conf. on Software Testing, Verification, and Validation (ICST)*, pages 220–229. IEEE Computer Society, 2008.