Build System with Lazy Retrieval for Java Projects

Ahmet Celik[1], Alex Knaust[1], Aleksandar Milicevic[2], and Milos Gligoric[1]
The University of Texas at Austin[1] (USA), Microsoft[2] (USA)
{ahmetcelik,awknaust}@utexas.edu, almili@microsoft.com, gligoric@utexas.edu

ABSTRACT

In the modern-day development, projects use Continuous Integration Services (CISs) to execute the build for every change in the source code. To ensure that the project remains correct and deployable, a CIS performs a clean build each time. In a clean environment, a build system needs to retrieve the project's dependencies (e.g., guava.jar). The retrieval, however, can be costly due to dependency bloat: despite a project using only a few files from each library, the existing build systems still eagerly retrieve all the libraries at the beginning of the build.

This paper presents a novel build system, MOLLY, which lazily retrieves parts of libraries (i.e., files) that are needed during the execution of a build target. For example, the compilation target needs only public interfaces of classes within the libraries and the test target needs only implementation of the classes that are being invoked by the tests. Additionally, MOLLY generates a transfer script that retrieves parts of libraries based on prior builds. MOLLY's design requires that we ignore the boundaries set by the library developers and look at the files within the libraries. We implemented MOLLY for Java and evaluated it on 17 popular open-source projects. We show that test targets (on average) depend on only 9.97% of files in libraries. A variant of MOLLY speeds up retrieval by 44.28%. Furthermore, the scripts generated by MOLLY retrieve dependencies, on average, 93.81% faster than the Maven build system.

CCS Concepts: Software and its engineering → Software notations and tools

Keywords: Build system; continuous integration service

1. INTRODUCTION

Continuous Integration (CI) [19], i.e., integrating all developer working copies into a shared mainline of source code as often as multiple times a day, has been widely adopted as part of agile software development [14] in general, and in extreme programming [13] in particular. In the environment of

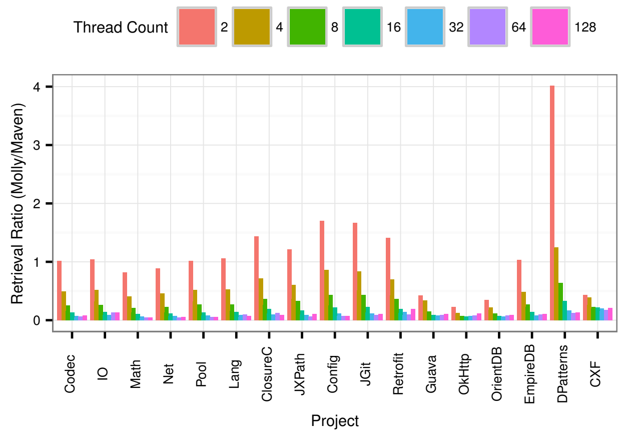

Figure 1: Comparison of a baseline dependency retrieval (the transitive closure is computed and libraries are retrieved in parallel, as implemented by Maven [9]) vs. the ideal case (all necessary files are known upfront and these files are retrieved in parallel). Values on the "Retrieval Ratio" axis are proportional to the baseline value (which is 1).

rapid code changes, CI was designed to prevent scenarios in which a developer working copy significantly diverges from the mainline, to the point where integration becomes difficult and time consuming. In the modern-day development, CI is typically performed by a dedicated service (*Continuous Integration Service*, CIS) after every push to the source repository. One such public service, Travis CI [60], for example, is used by more than 300K projects today, and performs approximately 130K builds daily [36].

To ensure that the project being developed remains deployable after every push, a CIS has to perform a *clean build* every time: starting from a clean state on disk, the latest copy of the source code is fetched from the repository, dependencies to other projects or libraries are retrieved, the project is compiled, and finally all tests are run.

In this paper, we focus on improving the *dependency retrieval* step. The common inefficiency of this particular step has been widely recognized [1,3,5,56,61,62]. (As an example, it can take up to an hour to retrieve all 1,701 libraries on which the Apache Camel project [6] depends.)

To motivate the problem further, consider Figure 1, where we take 17 popular open-source Java projects, all using the Maven [9] build system, and show how much faster, at least in the ideal scenario, the dependency retrieval step can be.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950358

The baseline (denoted as value 1 on the y axis) is the default dependency retrieval implemented by Maven: the transitive closure of libraries is computed first, then, the resolved libraries are retrieved in their entirety. (We tried configuring the size of the thread pool used by Maven for retrieving dependencies—by using the `-Dmaven.artifact.threads` option—but this change had no impact on the retrieval time.) In contrast, imagine if the full list of used libraries (e.g., `.jar` files) was known upfront—there would be no need for any additional computation, and the files could be retrieved massively in parallel. Further, imagine if not only the necessary libraries are known, but the exact individual files within those libraries (e.g., `.class` files)—only those files could be retrieved (still massively in parallel), saving both disk space and total retrieval time. For the 17 projects listed in Figure 1, we statically computed the minimal set of necessary files and measured the time it took to retrieve them in parallel (later in the paper we show how this can be achieved fully automatically by a general-purpose build system); when using 64 threads on a 4-core machine, the total retrieval time is reduced by 93.81% on average.

Problem: The key contributor to slow dependency retrieval is what we call *dependency bloat*. The *dependency footprint* of a project includes all libraries the project may possibly use. Build systems (e.g., [8, 9, 12, 32]) typically retrieve the entire footprint *eagerly*, prior to executing concrete build targets. In practice, however, often times not every library (or part thereof) gets used during a concrete build. Reasons are various: (1) a project may not use some of the transitive libraries (e.g., very few projects that depend on JUnit use JUnit's dependency Hamcrest), (2) the compilation phase needs only the APIs of classes rather than their implementation, (3) the compilation phase may fail, so there is no need to retrieve libraries used by any later phases (e.g., testing), and (4) even if all tests are successfully executed they may use only a small fraction of files from the libraries.

Solution: To mitigate the dependency bloat problem in a generic and application-agnostic way, we propose *lazy dependency retrieval*. Provided that the host language compiler/runtime allows for dynamic library loading, the build system ought to retrieve physical dependency artifacts (e.g., by retrieving them from a central repository) only when requested by the compiler/runtime, that is, only when triggered by execution of concrete build targets.

We also propose a variant of lazy dependency retrieval, where boundaries set by library providers are ignored by the build system, and instead, library content is retrieved on individual file basis. We dub this technique *elastic dependencies*. In our experiments, we observed that elastic dependencies reduce the total size of retrieved content by 89.80% on average.

Finally, to further reduce the cost of dependency retrieval for *subsequent* builds, we use our infrastructure to maintain a flattened list of resolved dependencies, called *transfer script*, which helps us get closer to the ideal retrieval time (Figure 1). Note that our infrastructure for lazy retrieval is still needed during the subsequent builds to retrieve new dependencies and update the flattened list for future builds.

Implementation: We implemented a prototype of the proposed technique in a novel build system for Java projects, called Molly. We implemented lazy dependency retrieval and elastic dependencies by modifying both the Java compiler and the Java runtime.

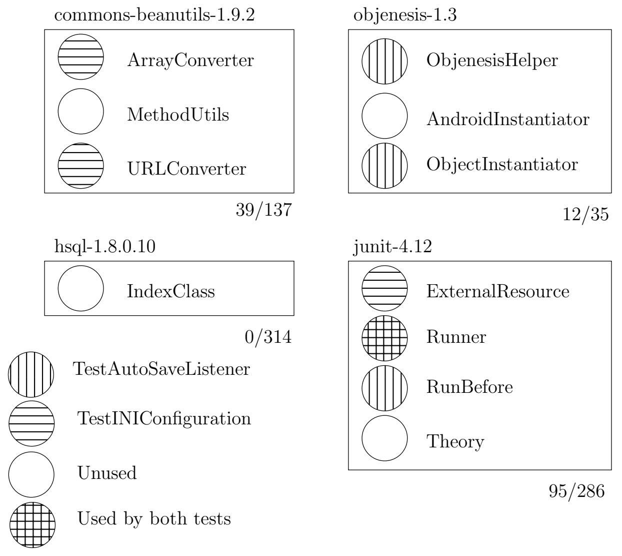

Figure 2: Dependencies for two tests in Config [7].

Evaluation: We evaluated Molly on 17 popular open-source Java projects. We report both empirical findings and concrete improvements observed when using Molly:

- (empirical) average time spent on dependency retrieval: 59.46% of the total clean build time;
- (improvement) average reduction in dependency retrieval time: 44.28% (due to lazy dependency retrieval);
- (improvement) average reduction in size of retrieved dependencies: 89.80% (due to both lazy dependency retrieval and elastic dependencies).

Molly achieves significant savings in both build time and disk space, which, we believe, can have a profound impact on reducing the cost of running and maintaining any CIS.

2. OVERVIEW

This section illustrates Molly through an example. First, we introduce a project to build. Next, we walk through the build process as carried out by both Maven (a widely used build system from Apache [9]) and Molly. We compare and contrast compilation and execution models of the two, and discuss the benefits brought by Molly. Finally, we describe the way Molly obtains a transfer script.

Example Project: `Config` [7] is an Apache project for parsing configurations from a variety of file formats. The `Config` project includes a Maven build script (`pom.xml`) listing all its dependencies. Each such dependency is a Maven project too, containing its own build script specifying its own dependencies. The most recent version of `Config` transitively depends on 269 libraries. Figure 2 depicts some of them as boxes containing a few representative classfiles (denoted as circles) and illustrates which of those classfiles are used by two randomly chosen unit tests.

Assume that a developer of `Config` pushes a change to the repository and a build is triggered by a CIS. We consider the following three scenarios: (1) compilation of test sources fails, (2) only one test is run, and (3) build succeeds. We next describe Maven and Molly and show how they manage dependencies for these three scenarios.

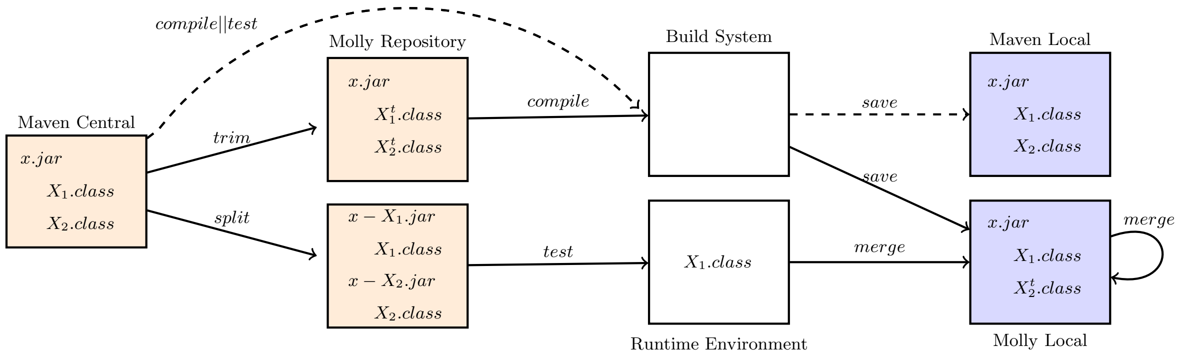

Figure 3: Molly's architecture (solid lines) and a conventional build system (Maven) (dashed lines). Maven eagerly retrieves *x.jar* prior to the compilation and test execution, whereas Molly lazily retrieves *trimmed* *x.jar* at compile time and lazily retrieves necessary executable code at runtime, *merging* when convenient.

Maven: Maven automatically manages a project's dependencies based on the user's specification in the build script. Specifically, Maven maintains the Maven Central Repository of publicly available libraries [10]. (We mined the Maven Central Repository and discovered that it currently includes 1,125,281 libraries, totaling 1,043GB.) Maven Central Repository allows library developers to autonomously deploy their libraries to the repository. Consequently, the library developers determine the size and content of the library, and consumers of the library have no way of specifying which parts of the library they need; they must use the entire library, or none of it.

When a build is executed, Maven automatically retrieves the necessary libraries for the executed target (based on `pom.xml`). The retrieved libraries are saved in the local repository on disk for later reuse. Note that if a project uses CIS, the libraries are removed from the local repository when the build finishes[1]. Figure 3 (dashed lines) illustrates the Maven build model. When a user invokes a target, Maven retrieves the dependencies *eagerly*, before the execution of the target begins.

For our example scenarios, importantly, Maven retrieves the same set of libraries *regardless* of which test is executed. The retrieved libraries include both the user specified dependencies and transitive dependencies with their entire contents. However, the failing compilation does not need the executable code of the compiled files, but only their public interfaces. The executed test `TestINIConfiguration` does not need dependencies (e.g., `ObjenesisHelper`) used by non-executed tests, and even a successful build does not need files that are never used (e.g., `IndexClass`).

Other popular build systems, including Ant + Ivy [8], Bazel [12] (limited), Gradle [32], and sbt [53] follow a similar approach with respect to dependency retrieval from the Maven Central Repository.

Molly: MOLLY introduces a novel execution model, shown in Figure 3 (solid lines), which tightly couples the build system, the compiler, and the runtime environment. Unlike the existing build systems, MOLLY retrieves parts of libraries lazily during the execution of the targets. Like other build systems, MOLLY maintains its own central repository (the two boxes that shown in Figure 3). The library developers deploy the same libraries as they would to Maven Central Repository. When a library is uploaded, MOLLY runs its *preexecution phase*, which *splits* the library into files and *trims* the library to reduce it to its public API. Specifically, MOLLY creates one library for each classfile in the original library (bottom second box from the left in Figure 3) and then trims classfiles from the original library, so that the trimmed library consists only of the public interfaces of the original classfiles (top second box from the left in Figure 3). For our running example, `Config`, MOLLY's splitting step will create 19,629 new libraries, and the trimming step will reduce the size of the libraries by 46.63%.

When a user invokes a target, MOLLY, in its *execution phase*, lazily retrieves only the classfiles that are needed for the execution and loads them into the main memory (third bottom box from the left in Figure 3). Then, either in parallel with the execution or offline (between two build runs), MOLLY *merges* retrieved classfiles into libraries in the local repository. (Note that merging is unnecessary for the execution of the build, but it is used to keep MOLLY transparent to the user). An additional advantage of MOLLY is that it makes the Molly Central Repository transparent to both the library developers (as they do not have to think how the best to split their libraries) and users of the libraries (as they do not have to worry about unnecessarily files being retrieved).

In our three scenarios for `Config`, MOLLY retrieves only the interfaces of classfiles needed to compile tests. If the user executes only `TestINIConfiguration`, MOLLY retrieves 39 of 137 files (the numbers are shown below the boxes in Figure 2) from `commons-beanutils`, and no files from either `Objenesis` or `hsql`. If the user instead executes `TestAutoSaveListener`, MOLLY retrieves 12 of 35 classfiles from `Objenesis`, but none from `commons-beanutils`, and also none from `hsql`. Even when the build succeeds, and all tests are executed, `Config` only uses 2,830/19,629 (14.42%) of classfiles. We present other results in Section 5.

During the build, MOLLY obtains a transfer script for the subsequent build; we discuss the details of the generated scripts in Section 3.3. As described earlier, transfer script contains the flattened list of files retrieved during the build. If the script exists, it is executed at the beginning of the build, and newly needed files are retrieved by MOLLY lazily during the build execution.

[1]Recently, several CISs, including Travis CI, introduced simple caching support that packages dependencies into an archive and uploads it to a remote storage service. However, retrieving a large cache can be costlier than retrieving the original dependencies [56]. Note that our work is complementary to caching and can reduce the cache size.

3. MOLLY PHASES

This section describes MOLLY's *preexecution* phase (performed when a library is deployed to Molly Central Repository, Section 3.1), *execution* phase (performed when a build target is executed, Section 3.2), and generation of the *transfer script* (Section 3.3). We assume the standard semantics for the Java classfiles [59] and the Java runtime [37]. Throughout this section we provide the intuition for why MOLLY preserves the behavior of the original build.

3.1 Preexecution: Trimming and Splitting

In the preexecution phase, a library, $\mathcal{L}$, is decomposed into a compile-time component ($\mathcal{L}^t$) and the runtime components ($\mathcal{L}^{\{r\}}$). For each Java classfile in the library, $z \in \mathcal{L}$, the compile time component of the classfile z^t (s.t., $z^t \in \mathcal{L}^t$ and $z^t = trim(z)$) should contain only the class's *public interfaces* (which we define later in more detail). Each runtime component $\mathcal{L}^r$ contains exactly one of the classfiles from the original library (i.e., $\forall z \in \mathcal{L}.\ \exists \mathcal{L}^r$ such that $z \in \mathcal{L}^r$ and $\nexists \mathcal{L}^{r\prime}$ such that $z \in \mathcal{L}^{r\prime}$).

3.1.1 Trimming Step

The objective of the trimming step is to decompose an existing library $\mathcal{L}$, a container of compiled Java classfiles, into a new trimmed library, $\mathcal{L}^t$, that can be substituted for $\mathcal{L}$ during the compilation phase of any dependent. Additionally, the *trim* function should extract a dependent-agnostic interface of $\mathcal{L}$ that is smaller, delaying decisions about which classes are really needed until runtime. The requirements of $\mathcal{L}^t$ are stated below:

1. For any project P that depends on $\mathcal{L}$, $\mathcal{L}^t$ may be substituted for $\mathcal{L}$ at compile time.
2. The product of the compilation with $\mathcal{L}^t$, will be identical to the product of the compilation with $\mathcal{L}$.
3. $\mathcal{L}^t$ should be the smallest possible library (in terms of code fragment) that satisfies the above two requirements and each classfile passes the Java verification.

Requirement 1 ensures that $\mathcal{L}^t$ is independent of a project P, so that $trim(\mathcal{L}) = \mathcal{L}^t$, which may be expensive, is performed only once per library. This requirement also guarantees that from the compiler's perspective, $\mathcal{L}^t$ is identical to $\mathcal{L}$, so that the developers of P can substitute $\mathcal{L}^t$ for $\mathcal{L}$ in compilation. The second requirement guarantees that will be possible to build P using $\mathcal{L}^t$ and later replace $\mathcal{L}^t$ with $\mathcal{L}$ at runtime. The third requirement sets the optimization goal for the trimming step: it should remove as much compiled code as possible, while still satisfying the first two requirements. If $\mathcal{L}^t$ satisfies these requirements, it will be a physically smaller version of $\mathcal{L}$, that can be transparently substituted for $\mathcal{L}$ at compile-time.

The trimming algorithm works by removing sections of the compiled classfiles that are not visible to its dependents. These can be broken up into two categories: *executable code* and *non-visible members*.

Rewrite rules: We define the *trim* function with a set of rewrite rules. Each rule has the following form:

$$C\ [before\ \rightarrow\ after]\ condition$$

where C is the context, $before/after$ are Java statements or class members (e.g., fields or methods) or the empty string (denoted by $\perp$), and *condition* defines the applicability of the rule. Interestingly, our rules apply in any context, so we always omit C. Additionally, we omit the condition if a rule always applies. *mod* denotes a set of Java modifies, $Expr^{Type}$ denotes any expression of the specified type, $Type$ denotes any type, and Pr denotes primitive and String types. We use f to refer to a field, m to refer to a method, and c to refer to a constructor. $\cdots$ are positional parameters and their use on the right side follow the order on the left side. Function $zero(Type)$ returns default value for the given Type (e.g., `null` for a reference type, 0 for `int`, etc.). We define a predicate $const(Expr)$ that holds iff the given expression is a constant expression (e.g., "string" for String type). The rewrite rules are shown in Figure 4. The class members and statements that are never explicitly used on the left side of the rules remain unchanged. Although our implementation of MOLLY trims Java classfiles, we define the rewrite rules on Java source level for the simplicity of exposition. (Note that some rules cannot apply to Java source code directly, and are mentioned below.) Most of the rules are straightforward, so we explain the intuition behind the rules on a high level and emphasize the rules that are potentially less obvious.

Private access members (fields, methods, constructors, and inner and nested classes) of $\mathcal{L}$ cannot be referenced by P at *compile time* due to access modifiers. Following the standard Java specification, the *trim* function may safely remove all *private* fields, methods, constructors, and classes of $\mathcal{L}$, since these may only be referenced in the classes in which they are defined. This is described by rules 2, 4, and 6. (Note that private members can be accessed through the Java reflection mechanism, but that belongs to the execution phase and we describe it in Section 3.2.)

$$static\ \{\cdots\} \rightarrow \perp \tag{1}$$

$$<mod>\ Type\ f\ \cdots \rightarrow \perp,\ \text{if}\ private \in mod \tag{2}$$

$$\begin{gathered}<mod>\ Type\ f\ = Expr^{Type} \rightarrow\\ <mod>\ Type\ f\ = zero(Type)\\ \text{if}\ private \notin mod\ \text{and}\ (Type \neq Pr\ \text{or}\ (Type = Pr\ \text{and}\\ (\neg const(Expr)\ \text{or}\ \{static, final\} \not\subseteq mod)))\end{gathered} \tag{3}$$

$$<mod>\ Type\ m(\cdots)\ \{\cdots\} \rightarrow \perp,\ \text{if}\ private \in mod \tag{4}$$

$$\begin{gathered}<mod>\ Type\ m(\cdots)\ \{\cdots\} \rightarrow\\ <mod>\ Type\ m(\cdots)\ \{return\ zero(Type)\}\\ \text{if}\ private \notin mod\end{gathered} \tag{5}$$

$$<mod>\ c(\cdots)\ \{\cdots\} \rightarrow \perp,\ \text{if}\ private \in mod \tag{6}$$

$$\begin{gathered}<mod>\ c(\cdots)\ \{\cdots\} \rightarrow <mod>\ c(\cdots)\ \{\}\\ \text{if}\ private \notin mod\end{gathered} \tag{7}$$

$$<mod>\ class\ M\{\cdots\} \rightarrow \perp,\ \text{if}\ private\ \in\ mod \tag{8}$$

Figure 4: Rewrite rules for the trimming phase.

$$\begin{gathered}<mod>\ Type\ m(\cdots)\ \{\cdots\} \rightarrow\\ <mod \cup native>\ Type\ m(\cdots);,\ \text{if}\ private \notin mod\end{gathered} \tag{9}$$

Figure 5: Alternative rewrite rule for 5 in Figure 4.

Package-private access members (fields, methods, constructors, and classes) are only accessible within the same package in which they are defined. Unfortunately, it is not possible to remove signatures of such members from $\mathcal{L}$ as a class C_1 of project P can be in the same package as C_2 of $\mathcal{L}$, in which case at compile-time, methods in C_2 should be able to reference package-private members of C_1. Since *protected* visibility implies *package-private* visibility, the trimmer also keeps the signatures of *protected* (and obviously, *public* members).

Bytecode instructions for any method and constructor can be replaced by a return statement with the appropriate default value (and no statement for the constructors and void methods). This is specified with rules 5 and 7. Note that performing the change on source code level would not be compilable in cases when a super class defines a constructor with non empty list of arguments. This is not an issue on the bytecode level, as exception would not be thrown until the class is instantiated. Figure 5 shows an alternative approach for trimming non-private methods: each method can be made `native`, indicating that its implementation is provided via a non-Java binary using the Java Native Interface (JNI), which can save several extra bytes per method.

There is a special treatment for non-private static final fields (rule 3). If a field has both `static` (not bound to object instances) and `final` (cannot be altered after initialization) modifiers and $const(Expr)$ holds for the expression assigned to the field, we do not rewrite the value of that field. For these fields, Java bytecode compilers may inline the values of "constant" fields during compilation. The problem could happen either in compilation phase (e.g., if those fields are used in a switch statement) or at runtime. Trimming static blocks (rule 1) is straightforward on bytecode level as the constant expressions have been assigned to the final static fields during the compilation.

Nested classes defined inside the scope of another class are no exception to the visibility rules described above. Anonymous classes, defined within the body of a method, are only accessible within the scope they are defined, and can be treated as private classes and safely removed from $\mathcal{L}$.

3.1.2 Splitting Step

Function $split(\mathcal{L})$ creates one $\mathcal{L}^r$ per classfile in $\mathcal{L}$. Currently, we do not consider the cases when a class C_1 depends on another class C_2 for every possible execution, and C_2 is used exclusively by C_1, (e.g., a private inner class can always be in the same $\mathcal{L}^r$ as the enclosing class). It is possible to perform static analysis to conservatively find further cases when one class always depends on another. Optimizing for these cases would complicate the lookup of retrievable classes at runtime and require additional consideration in the merge step; this optimization is left for future work.

It must be possible to determine the corresponding $\mathcal{L}^r$ from z^t at runtime so that it can be lazily resolved and correctly retrieved. Our solution to this is to concatenate the fully qualified class name with the unique identifier of the library, creating a name unique in both Maven Central Repository and in the Molly Central Repository. Additionally, libraries frequently contain other artifacts (e.g., images); we make these artifacts available in $\mathcal{L}^t$.

3.2 Execution: Retrieving and Merging

Although P may be compiled with the trimmed library $\mathcal{L}^t$, in order to execute P it is necessary to obtain the original executable code of $\mathcal{L}$. Requirement 2 of the trimming ensures that the compiled output of P using $\mathcal{L}^t$ is identical to the output of compilation with $\mathcal{L}$, so one option would be to fully replace $\mathcal{L}^t$ with $\mathcal{L}$ at run-time when any of the classes is needed. It may be the case that $\mathcal{L}$ is specified as a dependency, but never used (dangling, or only necessary for certain build targets). In this case this simple technique – replacement of $\mathcal{L}^t$ with $\mathcal{L}$ – would still be beneficial.

```
Input: $z$ the class attempting to be loaded
Input: $\text{Repo}_{\text{Local}}$ set of local dependencies
Input: $\text{Repo}_{\text{Remote}}$ set of remote split dependencies
Input: Mergeset class, dep. pairs to merge, initially $\emptyset$
 1 function DynamicLoad($z$, $\text{Repo}_{\text{Local}}$, $\text{Repo}_{\text{Remote}}$,
   Mergeset)
 2     $\mathcal{L} \leftarrow$ InitAndGetLibrary($\text{Repo}_{\text{Local}}$, $z$.name)
 3     if $z$.name $\notin \mathcal{L}$.replaced then
 4         $z' \leftarrow$ Retrieve($\text{Repo}_{\text{Remote}}$, $\mathcal{L}$.id, $z$.name)
 5         Mergeset $\leftarrow$ Mergeset $\cup$ $(\mathcal{L}, z')$
 6         return $z'$.src
 7     end if
 8     return $z$.src
 9 end function

10 procedure AsyncMerge(Mergeset, $\text{Repo}_{\text{Local}}$)
11     for all $(z, \mathcal{L}) \in$ Mergeset do
12         $\mathcal{L}' \leftarrow \mathcal{L} \setminus \{z^t\} \cup \{z\}$
13         $\mathcal{L}'$.replaced $\leftarrow \mathcal{L}'$.replaced $\cup$ $z$.name
14         $\text{Repo}_{\text{Local}} \leftarrow \text{Repo}_{\text{Local}} \setminus \{\mathcal{L}\} \cup \{\mathcal{L}'\}$
15         Mergeset $\leftarrow$ Mergeset $\setminus \{(z, \mathcal{L})\}$
16     end for
17 end procedure
```

Figure 6: DynamicLoad algorithm retrieves implementation of classes at runtime. AsyncMerge integrates the retrieved classes into the libraries on local disk concurrently, or at program end.

However, as shown in the motivating example, P will likely not require all classes of $\mathcal{L}$ to execute. Each execution on an input of P requires some subset of the classes of $\mathcal{L}$. Let $\mathcal{L}^i \subseteq \mathcal{L}$ be the classes required during execution of P on input i. Then we can represent the complete *partial dependency* of P on D as $\mathcal{L}^P = \bigcup_i \mathcal{L}^i$. Ideally P could be made to depend on $\mathcal{L}^P$ instead of $\mathcal{L}$. But due to dynamic dispatch, reflection, and other features of Java and other languages, it is challenging to compute $\mathcal{L}^P$ statically [17]. Instead we build $\mathcal{L}^P$ dynamically and incrementally, retrieving $\mathcal{L}^i$'s lazily during execution. This is the point where the build system, the compiler, and the runtime environment cross cut. In the first execution, the program is started with the trimmed libraries retrieved during the compilation. On subsequent executions, each library may be a mix of trimmed and executable classes loaded in prior executions.

3.2.1 Retrieving Step

In order to dynamically load the executable classes, all class loads must be intercepted. In Java this is possible by providing a system classloader, a JVM agent (which transforms all classes as they are loaded), or modifying the runtime environment. The DynamicLoad algorithm depicted in Figure 6 is called when the JVM attempts to load a class z. The algorithm first (line 2) obtains a local library that should contain z. The library may already be in the local repository if it was needed during the compilation (or a prior run). If the library is not available locally, a new library is created based on the build script; we assume that

```
Input: $\text{Repo}_{\text{Local}}$ set of local dependencies after build
Output: $\mathcal{S}$ executable script that retrieves dependencies
1 procedure GenScript($\text{Repo}_{\text{Local}}$)
2     for all $\mathcal{L} \in \text{Repo}_{\text{Local}}$ do
3         $\mathcal{S} \overset{+}{\leftarrow} \$fetch(`\mathcal{L}^t.id)\$$
4         $\mathcal{S} \overset{+}{\leftarrow} \{\$fetch(`\mathcal{L}.\text{id}, `z.\text{name})\$ | z.\text{name} \in \mathcal{L}.\text{replaced}\}$
5     end for
6 end procedure
```

Figure 7: GenScript algorithm generates the transfer script. The algorithm should be run after a clean build, but it can be extended to incrementally update the script.

the build script includes a mapping from a classfile to its library. Note that the algorithm ensures that the class is retrieved even if it was not used in compilation (e.g., accessed through reflection). Second (line 3), the algorithm checks if z has already been replaced with the executable class in a previous execution, and therefore does not need to be retrieved. Otherwise, if only the trimmed version of the class is present in the library or the class was not accessed previously, the underlying dependency resolution system is called to retrieve the $\mathcal{L}^r$dependency for z (line 4). This retrieved classfile is then marked to be merged into its library and returned to the JVM (lines 5 and 6).

3.2.2 Merging Step

After the executable version of a classfile is resolved and retrieved by DynamicLoad, it is desirable to cache the classfile for future executions. We perform the merge step to keep Molly transparent to the user. (If CIS performs clean builds, the merge step provides no value and it can be skipped.) AsyncMerge, described in Figure 6, performs this merging. For each `(class, library)` pair marked to be merged, AsyncMerge removes the trimmed classfile (if one exists) and includes the new classfile. Additionally, in order to distinguish between trimmed and original classes, a file inside of the library is updated to persist that the class has been replaced. These new libraries will then be loaded as $\text{Repo}_{\text{Local}}$ and used as input to DynamicLoad for the next execution of the program.

Performing AsyncMerge is expensive to execute, as the library file must be completely rewritten to add a new classfile. However, since the merged library is only relevant to the subsequent executions, the merging may be done in the background or postponed until JVM shutdown (which we implement). In Maven [9] and other build systems, the retrieved libraries may be shared between different programs, however synchronization is required to make this work with AsyncMerge. Finally, considering that class loading is invoked before any use of the class, under the assumption that the network connection is always available, Molly has no impact of the project's behavior.

3.3 Obtaining the Transfer Script

Prior sections described the core Molly features that are sufficient to build any Java project. Retrieving a single classfile at a time during the build execution, however, can slow down the build (Figure 1). The full power of Molly can be unleashed by generating a script that prefetches dependencies used in prior builds. Figure 7 shows the algorithm to generate the script. Note that the algorithm assumes that a clean build has finished previously. For each library that is in the local repository (after build execution), we insert $fetch$ statement for the trimmed version of the library, and one $fetch$ statement for each non-trimmed classfile. (We use standard notation from code generation community to represent code fragments and holes [39].)

On local machines or on CISs, which do not use dependency information to trigger builds and tests, the generated script $\mathcal{S}$ can simply be used prior to each execution and it can be updated as frequently as developers specify. Note that a stale script does not affect the correctness of the build, because Molly will retrieve newly added dependencies lazily and unused dependencies have no impact on the build.

More generally, $\mathcal{S}$ can be used as the complete list of dependencies of the project, which can be used to trigger build/tests [12, 21, 24, 30]. In this case, the script needs to be updated after each build to ensure consistency.

4. IMPLEMENTATION

This section briefly describes the implementation of our Molly prototype. (We only describe the implementation of the steps that are relevant for our evaluation.)

Trimming Step: Trimming is performed on the bytecode of classes within jar files using the ASM bytecode manipulation framework [11]. The trimmer walks the class file tree, applying the rewrite rules in Figure 4 to generate the class's trimmed bytecode, which is then added to a new trimmed jar. Non-classfiles in the jar are passed through untouched, and an additional file `JarSplitter.MOLLY_META` is added, containing the unique identifier of the artifact. The existence of this file is used during the retrieving step to determine whether the jar has been altered by Molly.

Splitting Step: Molly provides the `splitter` tool to split an existing jar into individual class dependencies. The tool creates a separate jar for each class, adding the `JarSplitter.MOLLY_META` file as before. Packaging each class into a separate jar allows for simpler integration with Maven, but these could simply be stored as separate classfiles or in a database. This step is transparently run when a library developer deploys their libraries to the Molly repository.

Retrieving Step: We modified OpenJDK's compiler and runtime (jdk8u74-b02) to implement this step. We currently implement two variants of Molly: (2) Molly^j, which retrieves trimmed jars lazily during compilation and retrieves the original jars during execution (a modification of the original technique described in Section 3.2), and (2) Molly^f, which retrieves trimmed files lazily during compilation and retrieves the individual files during execution (a modification of the original technique that uses the algorithm similar to Figure 6 both in the compilation and runtime).

We implemented our code in the exception handling blocks of the compiler, i.e., if a compiler cannot find a class, we intercept the exception and retrieve the appropriate library/file (if the library/file is available). For the runtime environment, we do a simple lookup to check if the local class is trimmed, and if so, the original class is retrieved. Molly^j and Molly^f currently retrieve jars and files sequentially.

5. EVALUATION

To assess the usability of Molly in the real world, we answer the following research questions:

RQ1: What is the average size difference between a Java library (jar file) and its public API (trimmed jar)?

Table 1: Details of evaluated open source projects.

Project	URL [https://github.com/]	SHA	KLOC	Jar #	File #	Jar Size [MB] Total	Jar Size [MB] Mean	Jar Size [MB] Max	Classfile Size [KB] Mean	Classfile Size [KB] Max
Codec	apache/commons-codec	e9da3d16	17.3	198	12896	24.93	0.13	1.93	2.14	31.83
IO	apache/commons-io	e8c1f057	27.2	211	13663	26.15	0.12	2.02	2.14	31.83
Math	apache/commons-math	471e6b07	174.8	266	19175	34.67	0.13	1.93	2.07	31.83
Net	apache/commons-net	4450add7	26.9	212	13678	26.20	0.12	2.02	2.15	31.83
Pool	apache/commons-pool	14eb6188	13.4	181	11484	21.97	0.12	1.93	2.09	28.03
Lang	apache/commons-lang	4777c3a5	66.0	218	14367	27.17	0.12	2.02	2.13	31.83
ClosureC	google/closure-compiler	831be0a9	254.8	152	11141	19.57	0.13	2.26	2.04	28.03
JXPath	apache/commons-jxpath	f1dde173	24.5	175	12244	22.50	0.13	1.93	2.19	31.16
Config	apache/commons-configuration	8dddebf1	64.3	269	19629	35.60	0.13	2.02	2.09	31.83
JGit	eclipse/jgit	070bf8d1	154.9	163	15793	25.84	0.16	2.90	2.02	24.36
Retrofit	square/retrofit	d26484c7	8.1	157	16060	34.88	0.22	12.95	1.95	24.36
Guava	google/guava	76e7d7a8	243.5	249	38991	90.08	0.36	41.82	2.30	72.39
OkHttp	square/okhttp	0cd6b186	47.0	177	17957	37.30	0.21	12.95	2.46	24.36
OrientDB	orientechnologies/orientdb	27e798b4	290.0	248	35132	59.93	0.25	7.28	2.21	24.90
EmpireDB	apache/empire-db	83c8fb2f	47.4	268	29989	55.00	0.21	3.20	2.21	62.66
DPatterns	iluwatar/java-design-patterns	4f56f7b0	16.5	334	54167	91.67	0.27	5.53	1.89	52.49
CXF	apache/cxf	f3185100	589.2	1003	205730	430.12	0.41	38.28	4.99	14202.00
Average	N/A	N/A	121.52	263.59	31888.00	62.56	0.19	8.41	2.30	868.57
Σ	N/A	N/A	2065	4481	542096	1063	3	142	39	14765

RQ2: What portion of the dependency retrieval time can be saved by lazy retrieval?

RQ3: What portion of the dependency retrieval time can be saved by using transfer script?

RQ4: What is the space reduction that can be achieved by lazy retrieval and elastic dependencies?

We use two machines for our experiments: (1) a 4-core 1.8GHz i7-4500U CPU with 8GB of RAM, running Ubuntu Linux 14.04LTS (which we will refer to as box^{c1}), and (2) a 4-core 2.7GHz i7 CPU with 4GB of RAM, running Ubuntu Linux 14.04LTS (which we will refer to as box^{c2}).

5.1 Projects Under Study

Table 1 shows the list of the projects used in our study, all open source, widely used, written in Java, and built with Maven. We selected these projects based on their popularity on GitHub, build system, and build outcome. Note that our experiments require projects that build successfully, in order to obtain all their dependencies. (Due to some test failures, which were mostly caused by flaky tests [27, 44], we exclude test execution for: Guava, OkHttp, OrientDB, and CXF.)

For each project, we show a short identifier[2], the project URL, the git commit SHA used in the experiment, and the number of lines of code (obtained using the `cloc` tool). Note that the different versions of the same library are treated as unique dependencies by Maven and are counted accordingly in the table. Additionally, we include statistics about the project's unique dependencies: the number of libraries on which the project depends (including transitive libraries), the number of files in these libraries, and the total, mean, and maximum size of each library. The set of dependencies also include all libraries that are used by Maven process during the build (e.g., plugins and their dependencies). The final two columns examine the Java bytecode files within the libraries, their mean and maximum size. Analysis of the libraries shows that the mean size of the libraries, and the sizes of classfiles within the libraries is independent of the project size. To confirm that sizes of libraries used by the selected projects are not unusual, we mined the Maven Central Repository, which includes 122,175 libraries (only the latest versions of libraries), totaling 110GB. Although, the selected projects use libraries whose size is somewhat higher than the median of libraries in the Maven Central Repository, the selected projects include no outliers.

[2]Projects are sorted based on the number of test dependencies, which is not included in the table.

5.2 Trimming Step

Table 3 shows the average size of trimmed libraries by applying the trimming algorithm (Section 3.1.1) to each project. We first compute the size of each original library ($size(\mathcal{L})$) and its corresponding trimmed library ($size(\mathcal{L}^t)$). Next, we compute the portion of $\mathcal{L}^t$ that is in $\mathcal{L}$, i.e., $Size[\%] = size(\mathcal{L}^t)/size(\mathcal{L}) * 100$. We then average $Size[\%]$ values across all libraries for each project and show these values in the table. The final row computes the average across projects. We can observe that the size of trimmed libraries differs by only few percentage points among the projects. Interestingly, we can see that the reduction in size is the smallest for `DPatterns`, which depends on classfiles with smallest size (Table 1, next to last column). In larger classfiles, bytecode instructions take more space than the constant pool, so the larger the classfile the more reduction may be obtained by trimming. We observed, by manually inspecting trimmed libraries, that a significant portion of the compiled bytecode is dedicated to the constant pool, which we do not alter.

Table 3: Average size of trimmed jars.

Project	Size [%]
Codec	55.26
IO	54.65
Math	53.86
Net	55.22
Pool	53.81
Lang	54.40
ClosureC	55.15
JXPath	54.45
Config	53.37
JGit	54.28
Retrofit	55.30
Guava	55.25
OkHttp	54.91
OrientDB	53.25
EmpireDB	54.61
DPatterns	57.02
CXF	53.42
Average	54.60

A1: Trimmed libraries are, on average, 45.40% smaller than the original libraries.

Table 2: **Comparison of Maven and Molly^j retrieval times. I_2—build time, including compilation and test execution, w/o dependency retrieval; $\Delta_I[s]$—dependency retrieval time; $\Delta_I[\%]$—percent of time spent on dependency retrieval.**

Project	box^{c1}					box^{c2}				
		Maven		Molly^j			Maven		Molly^j	
	I_2 [s]	Δ_I [s]	Δ_I [%]	Δ_I [s]	Δ_I [%]	I_2 [s]	Δ_I [s]	Δ_I [%]	Δ_I [s]	Δ_I [%]
Codec	10.85	48.61	81.76	14.36	56.98	13.19	64.52	83.03	17.86	57.53
IO	101.10	50.30	33.22	13.58	11.84	105.89	65.19	38.10	14.83	12.28
Math	129.19	66.30	33.92	15.78	10.88	142.41	99.71	41.18	22.75	13.77
Net	62.20	51.90	45.48	15.95	20.41	65.10	76.61	54.06	14.00	17.70
Pool	300.19	47.24	13.60	6.67	2.17	297.81	59.60	16.68	13.71	4.40
Lang	26.95	53.21	66.38	16.76	38.35	33.13	67.02	66.91	16.15	32.77
ClosureC	60.68	34.65	36.35	13.53	18.23	63.16	53.16	45.70	10.71	14.50
JXPath	8.28	45.10	84.49	9.64	53.79	10.47	56.55	84.38	10.99	51.22
Config	35.91	65.24	64.50	21.67	37.63	40.60	79.86	66.29	19.23	32.14
JGit	141.20	50.27	26.25	13.11	8.49	156.05	56.05	26.43	16.69	9.66
Retrofit	13.89	45.41	76.58	10.46	42.95	16.93	57.56	77.27	13.37	44.12
Guava	30.90	77.73	71.56	24.00	43.72	36.39	102.40	73.78	30.65	45.72
OkHttp	8.19	49.37	85.77	11.40	58.19	9.83	65.17	86.89	14.81	60.09
OrientDB	49.57	60.26	54.87	17.73	26.34	57.22	81.49	58.75	25.04	30.44
EmpireDB	16.63	70.73	80.96	23.77	58.83	18.89	94.50	83.34	24.68	56.65
DPatterns	97.51	87.93	47.42	31.46	24.39	108.40	117.38	51.99	38.13	26.02
CXF	215.09	266.21	55.31	119.26	35.67	249.80	318.09	56.01	156.75	38.56
Average	76.96	68.85	56.38	22.30	32.29	83.84	89.11	59.46	27.08	32.21
Σ	1308	1170	N/A	379	N/A	1425	1514	N/A	460	N/A

5.3 Retrieval Time

Setup: Instead of using the existing Maven Central Repository, we installed empty Maven and Molly repositories on a dedicated server exclusively available to us (denoted as box^s), to provide for fairness and stable measurements. We used the two machines described earlier as clients, which resulted in two configurations: $box^{c1} \leftrightarrow box^s$ and $box^{c2} \leftrightarrow box^s$. In both configurations, the bandwidth between the client and the server was 3.90 Mbits/sec (averaged over 10 repeated measurements using `iperf`).

To initialize the repositories on box^s, for each project we executed `mvn install` once without any changes to the project's build script. This command retrieved all libraries from the original Maven repositories, which we copied over to our private Maven repository. Additionally, we executed trim and split on each original library and added the resulting artifacts to our Molly repository.

First, we measure the time it takes to retrieve dependencies from box^s. (All retrievals from this point on are from box^s.) To do so, we measure Maven execution times of two separate runs: (I_1), where we delete the local Maven repository and execute `mvn install`, and (I_2), where we execute `mvn install -offline` without cleaning the local repository. I_2, thus, is the time spent building and executing tests, while $\Delta_I = I_1 - I_2$, is the time spent retrieving dependencies.

Second, we measure the time it takes to retrieve dependencies using Molly^j. Recall (Section 4) that Molly^j lazily retrieves trimmed libraries during compilation and lazily retrieves the original libraries during the execution. (In theory, running build with lazy retrieval could increase the build time if most of the libraries are retrieved, because Molly retrieves both the trimmed and original libraries. However, we did not observed a slowdown for any project, because many libraries are usually unused.) Exactly as we do for Maven, we run Molly twice (without and with a local repository present) to measure I_1 and I_2, and then compute Δ_I as the difference of the two. Note that I_2 is expected to be the same for both Maven and Molly (because they both perform exactly the same build), which our experiment confirmed.

Results: Table 2 shows the I_2 and Δ_I values for Maven and Molly^j when running on both of our client machines. The final two rows show the average and sum. We see that on both machines, Molly^j spends less time retrieving dependencies: 32.29% vs. 56.38% on box^{c1}, and 32.21% vs. 59.46% on box^{c2}. Based on these values, we compute the average saving of retrieval time: $1 - (32.29/56.38 + 32.21/59.46)/2$.

A2: Molly^j *saves, on average, 44.28% of retrieval time compared to Maven.*

5.4 Transfer Script Savings

In the introduction, we already described the ideal dependency retrieval time (Figure 1): 93.81% of retrieval time can be reduced on average by using transfer script (on box^{c2}; the results on box^{c1} are almost identical), which can be obtained from Molly^f. Note that the same figure shows that the initial retrieval time for Molly^f can be several times that of Maven due to the large number of HTTP requests, but that disadvantage would be gone by the end of the second build.

A3: Transfer script can save 93.81% of retrieval time.

5.5 Lazy Retrieval Disk Savings

Table 4 shows the savings in terms of the size of retrieved dependencies (and, therefore, disk space). The results are split into two main columns: Molly^j and Molly^f. For Molly^j, we show the total number of libraries retrieved by Molly^j (Jar #), and compare it with the total number of libraries retrieved by Maven (Jar [%]). (The numbers for Maven were reported in Table 1.) For Molly^f, we show the total number of retrieved files (File #), compare it with the total number of files in libraries retrieved by Maven (File [%]), and compute the reduction of the total size of retrieved artifacts when compared with Maven (Size [%]). Both Molly^j and Molly^f achieve consistent savings across all the projects. This particular metric is es-

Table 4: Stats for Jars and Files retrieved by Molly^j and Molly^f, respectively.

Project	Molly^j Jar #	Molly^j Jar [%]	Molly^f File #	Molly^f File [%]	Molly^f Size [%]
Codec	81	40.91	1198	9.29	9.75
IO	86	40.76	1299	9.51	10.09
Math	100	37.59	1545	8.06	8.74
Net	87	41.04	1272	9.30	9.92
Pool	67	37.02	1146	9.98	10.45
Lang	91	41.74	1350	9.40	9.97
ClosureC	52	34.21	1794	16.10	15.37
JXPath	71	40.57	1348	11.01	12.34
Config	120	44.61	2830	14.42	15.36
JGit	78	47.85	2060	13.04	14.24
Retrofit	51	32.48	1764	10.98	8.59
Guava	82	32.93	1176	3.02	2.52
OkHttp	50	28.25	434	2.42	1.56
OrientDB	95	38.31	1348	3.84	2.43
EmpireDB	185	69.03	3267	10.89	11.07
DPatterns	145	43.41	11161	20.60	20.25
CXF	425	42.37	15545	7.56	10.76
Average	109.76	40.77	2972.76	9.97	10.20
Σ	1866	N/A	50537	N/A	N/A

pecially important for distributed build systems [4,20], as it improves dependency caching [25,28,60]; we discuss this in more detail in the next section.

A4: Molly^f *uses only 9.97% of files used by Maven, and* Molly^j *uses 40.77% of jars used by Maven; the reduction in space with* Molly^f *is 89.80%, on average.*

6. DISCUSSION AND FUTURE WORK

Amortization: Although Molly provides the most benefits in CIS environments with clean builds without caching, Molly can bring benefits even when dependencies are (remotely) cached [2, 56]. Not only is Molly *orthogonal to caching*, but Molly may improve the performance of caches (particularly in a distributed environment), due to disk usage reduction. Additionally, if dependencies are cached, there is no need to cache them eagerly, but they can be cached when classfiles are lazily retrieved. This amortizes the retrieval time across several builds (of multiple projects).

It is important to mention that local caching (i.e., storing the cache on the machine that executed the latest build of the project) is non-existent in CISs. The reason may be obvious: CISs use the same machines to run builds of hundreds of thousands of projects (e.g., Travis CI is used by more than 300K projects [35,36]), and caching all dependencies for all projects on all machines is simply infeasible.

Transparency: Molly is transparent to both library developers and library users. Specifically, as only the necessary classes are retrieved, Molly incentivizes library users to make decisions about libraries based on the quality of each library instead of its packaged size. At the same time, library developers need not think about the sizes of their libraries. Additionally, library developers need not manually separate public APIs and its implementation, which is a non-trivial task.

Tracking: Maven Central Repository provides some statistics about libraries that it hosts. Molly could provide additional valuable information to the library developers, e.g., the list of classfiles that are most frequently used. The ability to track dependencies could be further used to optimize testing and verification [23,29,31,33,41,42,66] by tracking for each test/property what classes are being used.

Compression: We combined the pack200 jar compression tool [48], included with most JRE distributions, with the Molly's trimming algorithm. Combining both techniques reduces most projects' dependencies to 5-15% of their original size. Although combining Molly and pack200 may seem attractive, we did not proceed with this combination due to unpacking cost when the dependencies are retrieved, which would be done in the execution phase. Unlike unpacking, the packing could be done in the trimming step of the pre-execution phase. This step is performed either offline for each dependency as it is built or as it is uploaded to the repository, and hence is not a runtime overhead.

Future Work: We see several different ways to extend Molly, as well as a few opportunities for applying Molly's core lazy retrieval techniques in other domains. As for extending Molly, to save more space, Molly could track dependencies at a granularity finer than classfiles (e.g., methods); to improve the running time, Molly could speculatively prefetch classes and retrieve them in parallel (e.g., if class A is heavily used in methods of class B, Molly could start retrieving A as soon as B is requested). Beyond build systems, OS package managers could implement a similar approach to ours to retrieve parts of packages only upon request; the standard latex distribution on Linux systems is a good example, since it contains a large number of packages that most people never use.

7. THREATS TO VALIDITY

External: The reported results may not generalize beyond the projects used in our evaluation. To mitigate this threat, we chose active projects that differ in the application domain, the number of developers, the number of lines of code, the number of dependencies, and the number of authors. Additionally, several projects used in our experiments have been used in recent studies on regression testing and build systems [15,16,29,55].

The results for parallel retrieval time (Section 5.4) may differ based on the machine (e.g., HD instead of SSD) and network configurations. To mitigate this threat, we obtained results on two machine, which consistently showed improvement of the transfer script for retrieving dependencies over Maven's mechanism for retrieving dependencies.

Internal: Implementation of Molly and our scripts may contain bugs that may impact our conclusions. Molly's code base includes a number of tests that check the correctness of all phases of the proposed technique. We also manually inspected the outputs of Molly for several small and large examples.

Construct: For parallel retrievals, we used several values to initialize the thread pool, such that each value is a power of two. Our goal was to show the substantial speedup that can be obtained with our approach rather than to find the optimal value of threads in the thread pool.

In our evaluation, we used the default Maven configuration. As reported in Section 1, we also evaluated Maven retrieval time for different sizes of the thread pool, but we observed no savings; we used the same set of values for the thread pool size as for the evaluation of the transfer scripts.

Although a number of different build systems are available, we compared Molly only with Maven. Maven is still one of the most popular build systems for Java used by many open-source projects. In the future, we plan to further compare Molly with other build systems. Note however that every build system can benefit from lazy dependency retrieval and elastic dependencies.

8. RELATED WORK

This section describes prior work related to Molly.

Software Remodularization and Target Decomposition: Software remodularization is a closely related area of research attempting to cluster existing programs or dependencies into more meaningful modules. The Bunch [46] tool introduces using subsystem decomposition by graph partitioning of static inter-project file dependencies. Bunch provides different search metrics to allow for different tradeoffs between remodularization quality and performance. The Decomposer and Refiner tools [63] build upon this approach by forming the strongly connected components of an enormous file dependency graph [52], using expected dependent test triggers as a metric to partition the graph. Decomposer splits dependencies into only two subdependencies, and the chosen decomposition is tied to which dependents are available when the tool is run. Closely related work uses the strongly connected components of the file dependency graph to statically remove unused file dependencies from the build file within a project [67]. PoDoJA [18] seeks to minimize the download cost of code using dynamic information to statically repackage the downloadable jars. PoDoJA monitors the classes used in several execution scenarios, and optimizes the average download size across scenarios using a genetic-algorithm search for a partitioning of the classes within a jar. In comparison, Molly performs lazy dependency retrieval and elastic dependencies, which automatically reduce the dependencies between the projects: project depends only on the files from libraries that are used in one phase of the build execution. Our approach does not require any changes to libraries, and thus could help decoupling projects and reducing regression testing cost, without requiring substantial effort from developers.

Dependency Compression: Java packages code into jar files which use the zip format for easy distribution. In addition to standard compression algorithm such as gzip or bz2, specialized tools exist to shrink the size of jars. pack200 [48] performs a lossy compression on a jar file by joining the classes's constant pools, removing debug information, and applying the gzip algorithm. As discussed earlier, we experimented with combining pack200 and Molly, finding that while pack200 achieves on average a reduction to 30%, combining both approaches reduces to 5%-15%. Another tool, ProGuard [49] strips unused code from jars and obfuscates the rest, achieving between 10% and 88% reduction in jar size. Molly operates orthogonally to these compression tools, and combining the approaches achieves greater trimmed library compression at the cost of uncompressing overhead, which can be high for runtime deployment.

Build Systems: There are at least as many build systems as programming languages [9, 24, 26, 32, 53, 54]. Furthermore, studies show build system maintenance accounts for significant overhead of project development [4, 5, 45, 47, 65]. Over time, build systems have evolved to be integrated with a package manager that can resolve, retrieve, and install library dependencies automatically, similar to traditional package management software [58]. Although, resolution of dependencies with fixed versions is used by Maven, there is interest in taking advantage of a complex version resolution model [26, 63]. The Spack [26] build system leverages a constraint solver to determine the appropriate dependency version for large HPC software projects. Molly enhances these efforts by allowing the project to determine if changes in the dependency version are relevant. Other work has sped up Maven builds by delaying execution of the tests until all modules are built [16]. Our work is orthogonal, as we look how to optimize retrieval of dependencies regardless of the target being executed. Bazel [12] can create a library that includes only class APIs, however, the obtained classfiles cannot be loaded by the Java runtime environment as they do not pass the Java verification. This could be a limitation if Molly is implemented via the Java agent mechanism to avoid modifying the Java runtime environment.

Dynamic Patching: Molly builds on work on dynamic patching to insert executable code at runtime [50, 51, 57]. Java is designed to provide extensible dynamic loading [43]. Research in dynamic patching has shown that executable binary code may be patched at runtime with low overhead [34]. In a CORBA system [64], DLS implements a system and repository for loading code dynamically from the web at runtime [40]. DLS supports loading the appropriate implementation for the system, with fallback implementations.

Other Related Work: Java Web Start [38] is a tool for deploying code from the web that is included in Oracle's JRE. The newest versions of Web Start allow for lazy downloading of code at runtime, but only at the packaged jar level; these jars are downloaded when the application is started. Molly goes a step further by allowing lazy retrieval of individual files within the dependency, and integrating automatically into the project's build system. Static analysis of a partial program [22] requires inference of missing types to enable build of the partial program. Unlike work on analysis of a partial program, Molly lazily retrieves the full implementation of types used during the build.

9. CONCLUSIONS

We presented Molly, a new build system, which tackles the dependency bloat problem. Molly retrieves dependencies partially and lazily, i.e., only the necessary files, exactly before they are needed (while remaining fully automatic and completely transparent to the users). This is the key advancement over the existing build systems which enables Molly to: (1) reduce the retrieval time by 44.28%, (2) reduce disk space requirements by 89.80%, and (3) reduce retrieval time in typical cases by up to 93.81%. Molly excels in scenarios typical for builds on continuous integration services: the fast builds (due to transfer script) increase the overall throughput, and the small dependency footprint (due to lazy dependency retrieval and elastic dependencies) enables more efficient caching. We believe the cumulative effect of the savings achieved by Molly can make a significant difference for any continuous integration service, especially in the long run.

10. ACKNOWLEDGEMENTS

We thank the fellow students of EE 382V (Software Evolution) at The University of Texas at Austin for discussions on the material presented in this paper. We also thank Rajeev Alur, Nima Dini, Alex Gyori, Darko Marinov, John Micco, and Marko Vasic for their feedback on this work.

11. REFERENCES

[1] How not to download the Internet. http://blog.sonatype.com/2011/04/how-not-to-download-the-internet.

[2] WAD home page. https://github.com/Fingertips/WAD/.

[3] Your Maven build is slow. Speed it up! http://zeroturnaround.com/rebellabs/your-maven-build-is-slow-speed-it-up/.

[4] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. The evolution of the Linux build system. *Electronic Communications of the ECEASST*, 8:1–16, 2008.

[5] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible Java compiler. In *International Working Conference on Source Code Analysis and Manipulation*, pages 14–23, 2012.

[6] Apache Camel. https://github.com/apache/camel.

[7] Apache Commons Configuration. https://github.com/apache/commons-configuration.

[8] Apache Ivy. http://ant.apache.org/ivy.

[9] Apache Maven. https://maven.apache.org.

[10] Apache Maven Central Repository. http://search.maven.org.

[11] ASM home page. http://asm.ow2.org/.

[12] Bazel home page. http://bazel.io.

[13] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[14] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[15] J. Bell and G. E. Kaiser. Unit test virtualization with VMVM. In *International Conference on Software Engineering*, pages 550–561, 2014.

[16] J. Bell, E. Melski, G. Kaiser, and M. Dattatreya. Accelerating Maven by delaying test dependencies. In *International Workshop on Release Engineering*, pages 28–28, 2015.

[17] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *International Conference on Software Engineering*, pages 241–250, 2011.

[18] T. Bodhuin, M. Di Penta, and L. Troiano. A search-based approach for dynamically re-packaging downloadable applications. In *Conference of the Center for Advanced Studies on Collaborative Research*, pages 27–41, 2007.

[19] G. Booch. *Object Oriented Design: With Applications*. Benjamin/Cummings Pub., 1991.

[20] Build in the Cloud: How the build system works. http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html.

[21] M. Christakis, K. R. M. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *International Symposium on Formal Methods*, pages 643–657, 2014.

[22] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 313–328, 2008.

[23] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[24] S. Erdweg, M. Lichter, and W. Manuel. A sound and optimal incremental build system with dynamic dependencies. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 89–106, 2015.

[25] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. pages 11–20, 2016.

[26] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack package manager: Bringing order to HPC software chaos. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 40:1–40:12, 2015.

[27] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making system user interactive tests repeatable: When and what should we control? In *International Conference on Software Engineering*, pages 55–65, 2015.

[28] GitLab continuous integration. https://about.gitlab.com/gitlab-ci/.

[29] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[30] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 599–616, 2014.

[31] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Journal of Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[32] Gradle build tool. https://gradle.org.

[33] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, pages 332–358, 2003.

[34] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[35] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Automated Software Engineering*, 2016. To appear.

[36] It's Travis CI's 5th birthday, let's celebrate with numbers! https://blog.travis-ci.com/2016-02-05-happy-fifth-birthday-travis-ci.

[37] Java language and virtual machine specifications. https://docs.oracle.com/javase/specs.

[38] Java Web Start. http://www.oracle.com/technetwork/java/javase/javawebstart/index.html.

[39] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: Run-time code generation for Java and its applications. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 48–56, 2003.

[40] R. Kapitza and F. J. Hauck. DLS: a CORBA service for dynamic loading of code. In *On The Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, pages 1333–1350. 2003.

[41] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, 8(2):51–65, 1995.

[42] H. Kurshan, R. H. Hardin, R. P. Kurshan, K. L. Mcmillan, J. A. Reeds, and N. J. A. Sloane. Efficient regression verification. In *International Workshop on Discrete Event Systems*, pages 147–150, 1996.

[43] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. *SIGPLAN Notices*, 33(10):36–44, 1998.

[44] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[45] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17:578–608, 2012.

[46] B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch tool. *Transactions on Software Engineering*, 32(3):193–208, 2006.

[47] A. Neitsch, K. Wong, and M. Godfrey. Build system issues in multilanguage software. In *International Conference on Software Maintenance*, pages 140–149, 2012.

[48] Oracle. pack200 specification. http://docs.oracle.com/javase/7/docs/technotes/guides/pack200/pack-spec.html.

[49] ProGuard. http://proguard.sourceforge.net.

[50] M. Pukall, A. Grebhahn, R. Schröter, C. Kästner, W. Cazzola, and S. Götz. JavAdaptor: Unrestricted dynamic software updates for Java. In *International Conference on Software Engineering*, pages 989–991, 2011.

[51] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, 2000.

[52] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering*, pages 598–608, 2015.

[53] sbt - the interactive build tool. http://www.scala-sbt.org.

[54] SCons: A software construction tool. http://www.scons.org.

[55] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *International Symposium on Foundations of Software Engineering*, pages 237–247, 2015.

[56] Speeding up the build. http://docs.travis-ci.com/user/speeding-up-the-build.

[57] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: A VM-centric approach. In *Conference on Programming Language Design and Implementation*, pages 1–12, 2009.

[58] The Apt package manager. https://wiki.debian.org/Apt.

[59] The class file format. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html.

[60] Travis CI - test and deploy with confidence. https://travis-ci.com/.

[61] Travis CI - issue 1441. https://github.com/travis-ci/travis-ci/issues/1441.

[62] Travis CI - issue 189. https://github.com/yegor256/thindeck/issues/189.

[63] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni. Automated decomposition of build targets. In *International Conference on Software Engineering*, pages 123–133, 2015.

[64] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine*, 35(2):46–55, 1997.

[65] X. Xia, X. Zhou, D. Lo, and X. Zhao. An empirical study of bugs in software build systems. In *International Conference on Quality Software*, pages 200–203, 2013.

[66] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[67] Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing false code dependencies to speedup software build processes. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 343–352, 2003.