September 13, 2018

Context-Aware Software Builds

Karthik Ravi Shankar

Context-aware software builds

ABSTRACT

Software libraries are of the static or dynamic variety. In a static library, code from the library is integrated into the executable at compile time. The resulting executable is relatively large but runs fast and in a stand-alone manner. In a dynamic library, code from the library is linked to the executable at run-time. The executable is smaller, and due to the sharing of dynamic libraries across processes, has less memory overhead. However, running the executable is contingent on the presence of the dynamic library in the machine that it runs on. Linking a library at run-time can also cause loss in speed.

This disclosure presents machine-learning based techniques to optimally identify a build target as a shared or static library. A recommendation is made to the software developer regarding an optimal setting (dynamic or static) for compilation. The techniques enable a developer to make informed design decisions.

KEYWORDS

Software library; shared library; static library; dynamically linked library; makefile; build system; build tool; dynamic library; directed acyclic graph; depth-first search; executable

BACKGROUND

Build tools are software packages that automate the process of program compilation. Build tools are essentially functional languages that map a set of source files to a target executable. Build tools typically assume that build actions are idempotent, e.g., an invocation of a build command with the same input and options will create the same output. This assumption enables a build tool to memorize previously performed actions, and only perform build actions on files that have changed, leading to efficient creation of updated executables.

While their primary goal is to efficiently create executables, build tools often try to meet a variety of secondary goals, which at times might be at odds with each other. An example is the linking of software libraries to executables. Libraries are of static or dynamic variety. In a dynamic library, also known as shared library and identified, for example, by files with extension `.so`, `.dll`, or `.dylib`, code from the library is linked to the executable at run-time. Code relating to the library is in the `.so` (or `.dll` or `.dylib`) file, and is referenced by programs that use the library at run-time. A program using a shared library makes reference to just the code that it uses in the shared library. In a static library, identified typically by files with `.a` or `.lib` extensions, code from the library is linked to the executable at compile time. A copy of the code from the static library program utilized by a program is made part of the program.

Dynamic (shared) libraries enable replacement of a shared object with one that is functionally equivalent, and have performance benefits without needing recompilation of the program that makes use of it. Shared libraries are loaded into an application at run-time, and hence hew to the general mechanism for implementing binary plug-in systems. Compared to static linkage, the executable that calls a shared library is smaller. Due to the sharing of dynamic libraries across processes, such executables have less per-executable memory overhead. However, running the executable is contingent on the presence of the dynamic library on the machine. Shared libraries also take longer for executing functions, and can have a run-time loading cost as well, as symbols in the library need to be connected to the objects they use.

Use of static libraries can result in a larger size of the binary executable; however, such executables can run in a stand-alone manner without the presence of the library on the machine. As static libraries are linked at compile time, there are no run-time loading costs.

Thus, dynamic and static libraries each have their respective advantages and disadvantages. In particular situations, e.g., where speed is more important that memory usage, or where an executable is expected to run in a stand-alone mode without access to libraries, statically linked libraries are advantageous. In situations where memory is at a premium or the size of the binary executable is to be optimized, dynamic linking of libraries is advantageous. Traditionally, the decision to statically or dynamically link a library is driven by human or developer intuition, which can be sub-optimal, or prone to bias.

DESCRIPTION

This disclosure describes machine-learning techniques to statically analyze build files and identify build targets to be declared as static libraries, shared libraries, or executables.

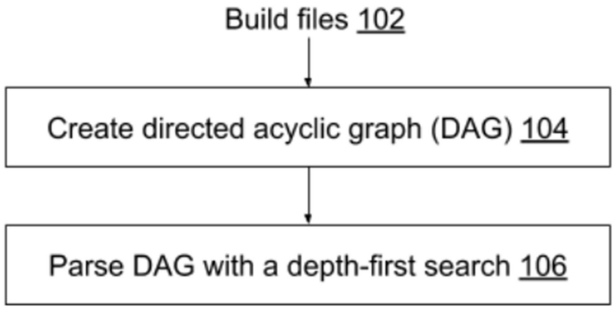

Fig. 1: Identifying build targets as static libraries, shared libraries, or executables

Fig. 1 illustrates identifying build targets as static libraries, shared libraries, or executables, per techniques of this disclosure. A plug-in within the build tool accepts as input build files (102), e.g., *make* files, *ant* files, etc., and creates a directed acyclic graph (104) of the targets and their dependencies. The plug-in accounts for the version of the targets being built. The directed acyclic graph is parsed with a depth-first search (DFS) (106) that enables enumeration of the inbound dependencies of targets. Targets with a relatively large number of

inbound dependencies are generally recommended to be shared libraries, while targets with a relatively small number of inbound dependencies are recommended as static libraries. Targets which are being updated with new revisions and/or with known backward compatibility issues are marked as static libraries.

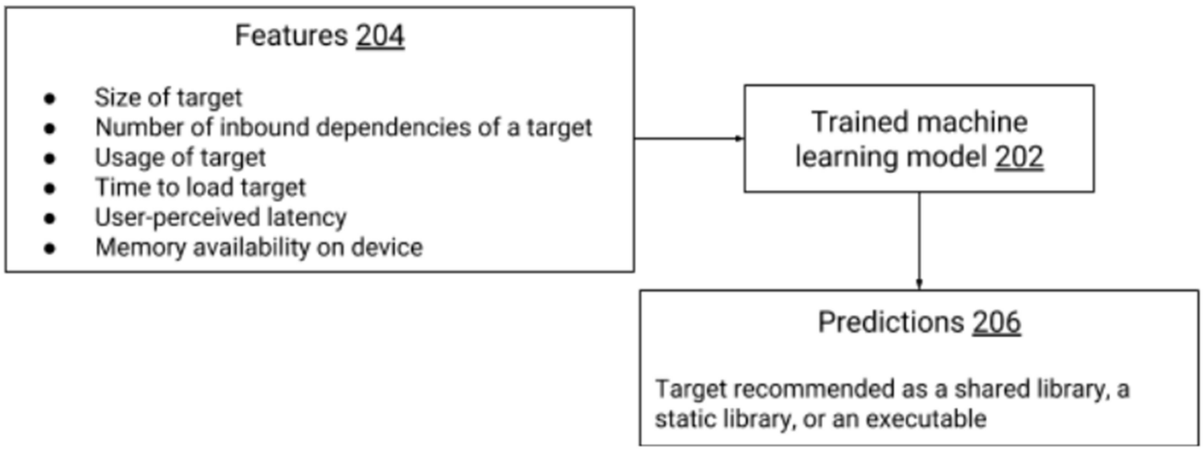

Fig. 2: Machine learning to classify targets

The DFS uses advantageously machine learning techniques, as illustrated in Fig. 2, to classify targets as static or dynamic libraries. A machine-learning model (202) accepts as input a number of features (204) of the build target and makes predictions (206) relating to the target. Features accepted as input include, for example, the size of the target, the number of inbound dependencies of the target (e.g., number of other targets dependent on the target), usage of these targets (e.g., how often a particular target is loaded into applications), time to load the target, user-perceived latency, memory availability on device, etc. Predictions of the machine-learning model include a recommendation to the developer regarding whether the target is best utilized as a shared library, a static library, or an executable.

The trained machine learning model can be, for example, a regression learning models, utilize neural networks, etc. Example types of neural networks include long short-term memory

(LSTM) neural networks, recurrent neural networks, convolutional neural networks, etc. Other machine learning models, e.g., support vector machines, random forests, boosted decision trees, etc., can also be used. Reinforcement learning can be used to advantageously incorporate past predictions as feedback to improve machine-learning performance. The machine-learning model is trained with known pairs of input features and optimal predictions.

In this manner, the techniques of this disclosure optimally select a library to be statically or dynamically linked depending on the software environment. Code duplication can thereby be reduced, and the size of the binary executable or image is optimized. This is advantageous while updating the image, e.g., via over-the-air download. In addition, optimized image size conserves the battery of a mobile device. Further, the techniques create a single, central repository that stores and maintains targets and dependencies within the software build, which is useful in software projects of any size, but especially in large-scale projects. The techniques offer a global view of the software project, which in turn enables a quick re-build of different parts of the system when a dependency changes.

CONCLUSION

This disclosure presents machine-learning based techniques to optimally identify a build target as a shared or static library. A recommendation is made to the software developer regarding an optimal setting (dynamic or static) for compilation. The techniques enable a developer to make informed design decisions, e.g., whether to include the library in a static manner such that the executable includes the library, or in a dynamic manner, such that the library is linked to the executable at run-time.