AMBuild

AMBuild is a tool for building software projects and creating release packages. It is targeted at C++ projects, though it can be used for anything. It has been tailored to solve three major problems that most build tools do not address:

- **Accuracy**. You should *never* need to clean a build. Clean rebuilds are unnecessary and a waste of your time. AMBuild always computes minimal rebuilds accurately - any failure to do so is considered a bug.
- **Speed**. Most build systems need to traverse the entire dependency graph for changes. AMBuild only needs to look at the set of changed files on the filesystem.
- **Flexibility**. Build scripts are written in Python, so they're very easy to write and provide full programmatic control.

Keep in mind, AMBuild is neither widely used nor has it been used on a wide variety of systems. AlliedModders has used it successfully for small to medium-sized C++ projects on Linux, Mac, and Windows. Our largest project, SourceMod, has 700 C++ files and over 200,000 lines of code. We're happy to receive feedback on GitHub if you use it in your own projects.

AMBuild 2.2.1 is the latest release. It supports cross-compiling, multi-architecture builds, and precompiled headers.

Contents

Motivation

AlliedModders C++ projects require a lot of customization. The set of flags passed to the compiler, their order, how things get linked, is all delicate and complicated. In addition, each Source game requires different linkage, so our C++ files usually get compiled multiple times, over and over again, into separate binaries. Lastly, our projects are large, and minimizing build time through correct dependency computation and parallelization is important. Very few build systems can handle any of these scenarios well, much less all of them, so we sought to make a new build system.

The initial version of AMBuild only solved flexibility problems. By controlling the build pipeline through Python, we were able to generate 15+ different binaries from the same source files without any code duplication. Over time the AMBuild script syntax has become very simple; you no longer need a complex project to justify AMBuild.

The modern version of AMBuild (also known as AMBuild 2), is modeled after Tup (http://gittup.org/tup/). Tup is a huge advance forward in build systems, and it is likely that one day AMBuild will simply use Tup as a backend. If you are looking at AMBuild as a build platform for your projects, you may want to see whether Tup meets your needs instead.

Requirements

Python 3.3 or higher is required.

Additionally, pip and setuptools (both Python components) are also required. If your distribution or Python installation does not provide for these, see Installing Pip (https://pip.pypa.io/en/stable/installing/) for manual instructions.

AMBuild has been tested to work on the following platforms. Although builds of Python for specific architectures were tested, we expect that other architectures will work as well.

- Windows 7+ with x86, x86_64, arm, or arm64 targets.
- Linux with clang and GCC.
- OS X 10.9+ with x86 and x86_64 targets.

It has also been tested to work with the following compilers:

- Visual Studio 2015+
- GCC 4+
- Clang 3+
- Emscripten 1.25+ (JS output mode only, lib deps do not work)

Installation

AMBuild can be downloaded via GitHub at https://github.com/alliedmodders/ambuild (https://github.com/alliedmodders/ambuild).

```
$ git clone https://github.com/alliedmodders/ambuild
$ pip install ./ambuild
```

Note: By default, pip will perform per-user installations. To install globally for all users, you will need to use "sudo" (or run as Administrator on Windows).

Tutorial

See the AMBuild Tutorial for more information.

API

See the AMBuild API article for more information.

Technical Overview

AMBuild is separated into a *frontend* and a *backend*. The frontend is responsible for parsing build scripts (this is known as the *configure* step). The backend is responsible for actually performing builds. The frontend and backend are separate, and it is possible to use a different backend other than AMBuild. For example, there are plans for the configure step to be able to produce Visual Studio project files.

Configuring

Build scripts are written in Python, and they are parsed whenever you configure the build. If a build script changes, it will automatically re-trigger the configure process on the next build. When a configure takes place, any files left by an old build are removed if they are modified or removed in the new dependency graph.

The details of the generated dependency graph are stored in an SQLite database, which is located in a hidden `.ambuild2` folder inside the build path.

Building

The build process involves a few important steps that are executed every time you use the `ambuild` command:

1. **Damage Computation**. Builds a list of all files that have changed since the last build. This is based on filesystem timestamps, and either a forward or backward time change is enough to be considered "damaged" (or dirty). For example:

```
$ ambuild --show-changed
/home/dvander/alliedmodders/mmsource-central/loader/loader.cpp
```

2. **Partial DAG Construction**. A partial dependency graph is built based on the list of damaged files. This is the entire set of nodes in the graph which need to be recomputed. The output of this step can be seen with `--show-damage`. For example:

```
$ ambuild --show-damage
 - package/addons/metamod/bin/server_i486.so
  - cp "../loader/server_i486/server_i486.so" "package/addons/metamod/bin/server_i486.so"
   - loader/server_i486/server_i486.so
    - c++ loader.o gamedll.o serverplugin.o utility.o -m32 -static-libgcc -shared -o server_i486.so
     - loader/server_i486/loader.o
      - [gcc] -> c++ -Wall -Werror -H -c /home/dvander/alliedmodders/mmsource-central/loader/loader.cpp -o loader.o
       - /home/dvander/alliedmodders/mmsource-central/loader/loader.cpp
```

3. **Task Construction**. The partial DAG is simplified into a tree of commands to run. The output of this step can be seen with `--show-commands`. For example:

```
$ ambuild --show-commands
 - cp "../loader/server_i486/server_i486.so" "package/addons/metamod/bin/server_i486.so"
  - c++ loader.o gamedll.o serverplugin.o utility.o -shared -o server_i486.so
   - [gcc] -> c++ -Wall -Werror -H -c /home/dvander/alliedmodders/mmsource-central/loader/loader.cpp -o loader.o
```

4. **Updating**. Each task in the task tree is executed and the results are processed to either reject the build, or update the state of the dependency graph. At this phase, tasks are executed in parallel based on the number of CPU cores available. The task graph is also shared to maximize throughput. You can see the sequential build steps with `--show-steps`:

```
$ ambuild --show-steps
task 0: [gcc] -> c++ -Wall -Werror -H -c /home/dvander/alliedmodders/mmsource-central/loader/loader.cpp -o loader.o
  -> loader/server/loader.o
task 1: c++ loader.o gamedll.o serverplugin.o utility.o -m32 -static-libgcc -shared -o server.so
  -> loader/server/server.so
```

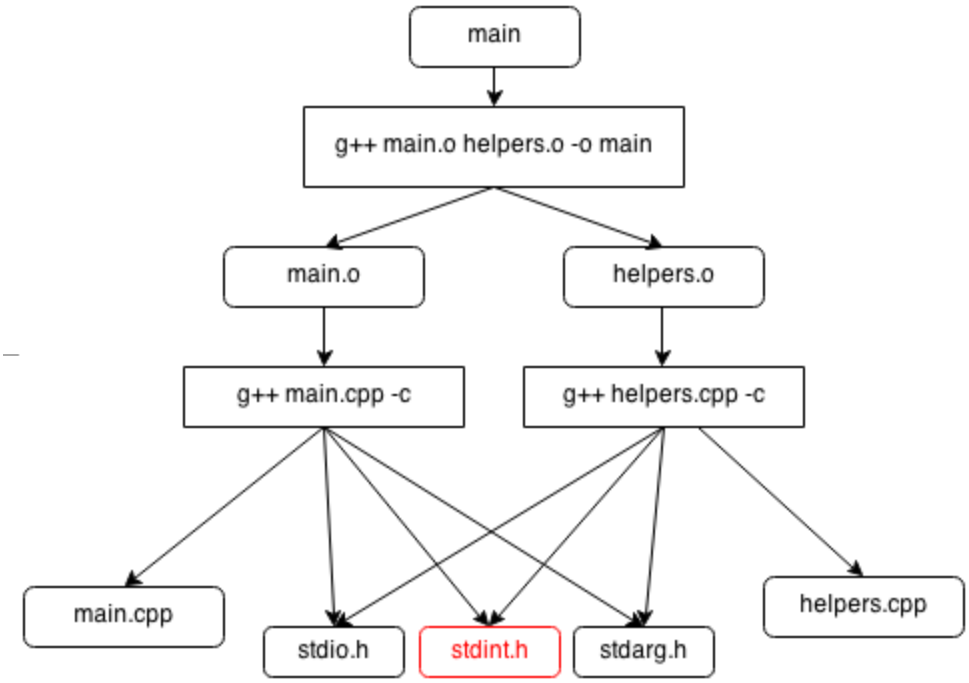
```
task 2: cp "../loader/server/server.so" "package/addons/metamod/bin/server.so"
  -> package/addons/metamod/bin/server.so
```

Maximizing throughput in Python is tricky since it has limited capability for IPC and multithreading. AMBuild spawns worker processes based on the number of CPUs available, which run task jobs and return the results.

Comparisons

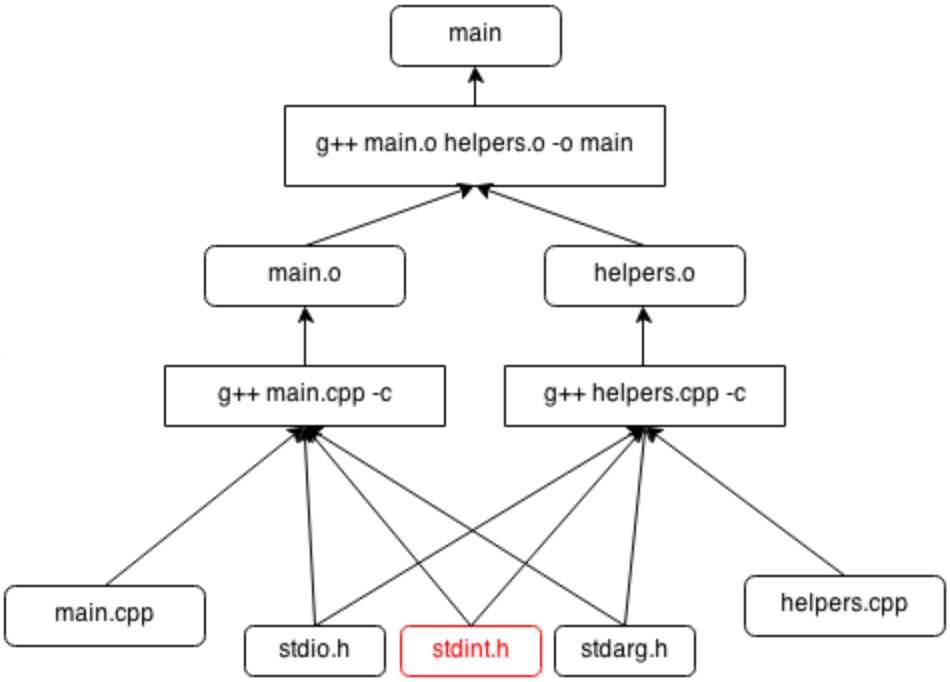
Make, Speed

Make uses a recursive update scheme. Starting with the top-level rule, Make will recursively search all dependencies to find outdated rules, and it will update all rules as it unwinds. This usually involves touching way more rules than are necessary. Consider the following dependency graph:

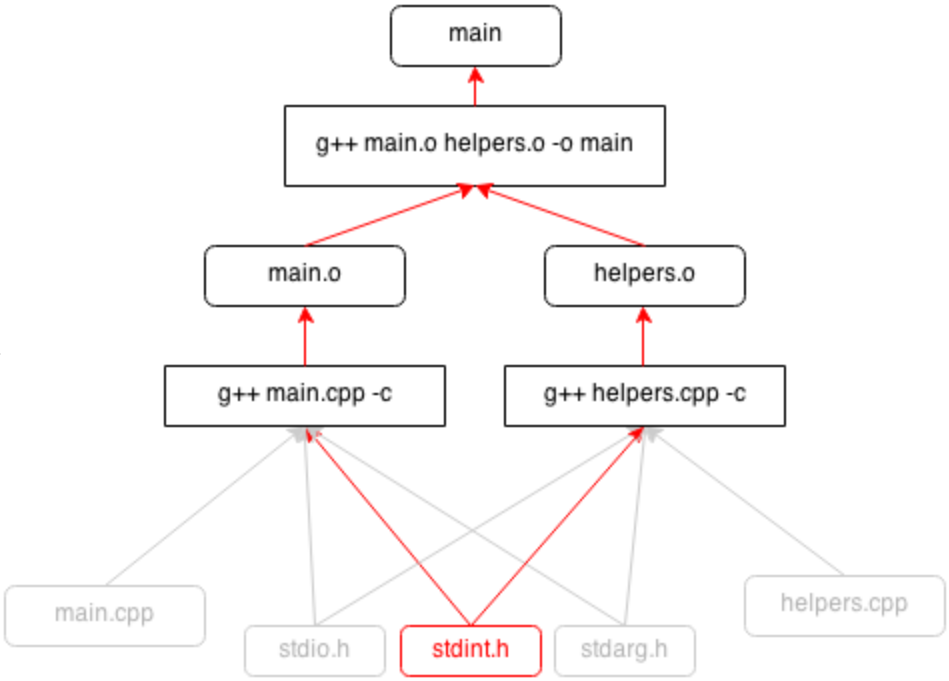


In this graph, `stdint.h` has been modified. Even though it's the only rule that has been changed, Make must visit every single node in the graph to discover that it has changed. Furthermore, it has to visit it multiple times: once for `main.o`, and another time for `helpers.o`. In a large dependency graph, this is very expensive.

In AMBuild, the direction of the graph is reversed:



With this graph, it is possible to visit every node once. Once we know that `stdint.h` has changed, we can follow the edges upward and ignore everything in the graph that is unaffected:



Make, Accuracy

AMBuild strives to be accurate by design, in two ways:

- Computation of dependencies should be as minimally accurate as possible. Changing a build script should not result in a complete rebuild.
- An incremental build should be exactly the same as a fresh build.

It is difficult to achieve very accurate rebuilds in Make. For example, consider a Makefile that lists source files and CFLAGS, and invokes the C++ compiler to build and link them:

```
# Makefile
CFLAGS = -Wall

helpers.o: helpers.cpp
  $(CC) $(CFLAGS) helpers.cpp -c -o helpers.o
main.o: main.cpp
  $(CC) $(CFLAGS) main.cpp -c -o main.o

main: main.o helpers.o
  $(CC) main.o helpers.o -o main
```

If you change `helpers.cpp`, you will get a minimal rebuild. If you add a new source file, you will likely get a minimal rebuild. However, if you *remove the `helpers` rule completely*, the build will be incorrect, because `Make` cannot tell that `main` needs to be relinked.

One way to solve this is to have certain rules, like the link rule, have a dependency on `Makefile` itself. But then if you change `CFLAGS`, it will trigger a relink, even though that rule does not depend on `CFLAGS`. You would need to break the Makefile down into many smaller Makefiles.

AMBuild mitigates these problems by intelligently updating the dependency graph. If a reparse of the build scripts produces identical nodes, it will not consider those nodes as stale.

Furthermore, when deleting rules, Make will leave their outputs behind. There is not an easy way to solve this. Leaving old outputs behind is undesirable for a number of reasons. Stale outputs might fool a user or developer tool into thinking the build has succeeded, when in fact it has failed. Stale outputs might be loaded accidentally, i.e. `LD_LIBRARY_PATH` picking up a library that should not exist. Tests that developers forgot to update might pass locally because of a stale output.

In AMBuild, leaving stale files behind is an error, since they would not be produced by a fresh build.
