Build System: bb

Mark Boorer • 31.03.2021

bb Overview

Why

- What problems does it solve

How

- High level overview
- Package descriptions
- Target definitions
- What does it look like to interact with from a developer perspective

Live Demo

Q&A

Why?

Manual Software building is painful

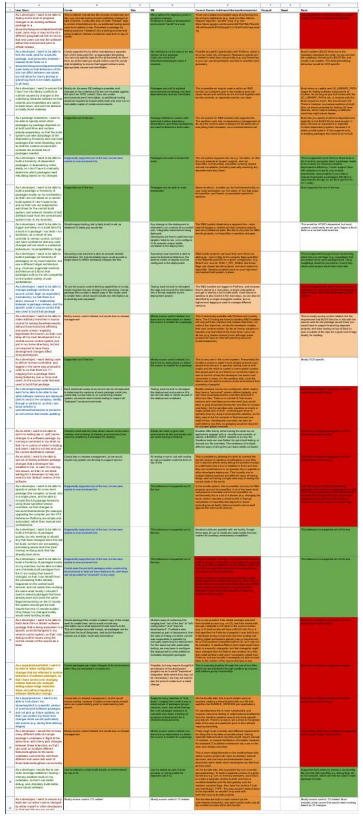
- Often requires patches that need to keep up with upstream changes
- Runtime is potentially unstable (incorrect LD_LIBRARY_PATH or RPATH)

Other systems aren't feature complete

- NixOS exposes only a single build tree
- REZ's resolver requires manual developer tuning
- Conda is slow and relies on patching binaries for $ORIGIN lookups

Extensive investigation of existing solutions

- ~100 user stories from various departments (R&D, PipeTD, Prod)
- Main focus on bb, Bazel, Buck, Conda, Rez, and our current system



How?

High Level Overview

- Package build instructions are described via executable / scripted code (js/lua/?) and ran in chrooted isolation
- The collection of package build instructions for the entire company is maintained as a single git repo - the “deployment”
- The deployment also contains the list of final execution environments (targets) that we are using (maya, zeno, nuke, etc)
- Developers interact with the deployment to build new environments, update existing ones, all in isolation. No more testing in production

How?

High Level Overview

- Merges to the deployment are predicated on all targets building successfully, and all unit/integration tests passing
- Built packages are stored in immutable, hashed locations
- Build artefacts are re-used when targets have overlapping, binary identical dependencies and only diverge when necessary
- Build artefacts can be stored in local or remotely shared locations, increasing the opportunity for artefact re-use and therefore decreasing build times

How?

Package Descriptions

```
function openexr(ctx, params){
    set_build_environment(ctx, "cpp_toolchain", "ilmbase", "zlib");
    get_source(ctx, params, "openexr");
    do_autotools_build(ctx, {"prefix":"/usr"});
    set_runtime_environment(ctx, "ilmbase", "zlib");
}
registry.register_fn("openexr", openexr);
```

```
function openimageio(ctx, params){
    set_build_environment(ctx, "cpp_toolchain", "boost", "openexr", "libjpeg",
                               "libpng", "libtiff", "libraw", "robin_map");
    // oiio's final filename differs from the url
    var filename = format("oiio-Release-%s.tar.gz", params['version']);
    get_source(ctx, params, "openimageio", {"outfile": filename});

    //robin_map is provided in its own package
    var cmake_extra_args = "-DBUILD_MISSING_ROBINMAP=OFF";

    // Disable python bindings for now (needs pybind11)
    cmake_extra_args += " -D USE_PYTHON=OFF";

    // Hack to work around boost 1.70.0 and cmake
    var boost_params = ctx.lookup(new P("boost"));
    if (boost_params['version'] == "1.70.0"){
        cmake_extra_args += " -DBoost_NO_BOOST_CMAKE=ON";
    }
    do_cmake_build(ctx, {"extra_args":cmake_extra_args});
    set_runtime_environment(ctx, "cpp_toolchain", "boost", "openexr", "libjpeg",
                                 "libpng", "libtiff", "libraw");
}
registry.register_fn("openimageio", openimageio);
```

```
function findutils(ctx, params){
    set_build_environment(ctx, "c_toolchain", "sed", "grep", "gawk");
    get_source(ctx, params, "findutils");
    ctx.shell("sed -i 's/test-lock..EXEEXT.//' tests/Makefile.in"); // remove broken test

    // fixes for glibc 2.28+
    ctx.shell("sed -i 's/IO_ftrylockfile/IO_EOF_SEEN/' gl/lib/*.c");
    ctx.shell("sed -i '/unistd/a #include <sys/sysmacros.h>' gl/lib/mountlist.c");
    ctx.shell('echo "#define _IO_IN_BACKUP 0x100" >> gl/lib/stdio-impl.h');

    do_autotools_build(ctx, {
        "prefix":"/usr",
        "configure_args":"--localstatedir=/var/lib/locate"
    });
    ctx.shell("mv -v /usr/bin/find /bin"); // move resulting binary to /bin
    ctx.shell("sed -i 's|find:=${BINDIR}|find:=/bin|' /usr/bin/updatedb");
    set_runtime_environment(ctx, "c_toolchain");
}
registry.register_fn("findutils", findutils);
```

How?

Package Descriptions

- Are programmable code, so repetitive patterns can be factored out into functions
- Allow for complex logic to be expressed (such as V.01 uses autotools, V.02 needs CMake, etc)
- Could be expanded to parse definitions from static files as well
- Are small and fast enough to be executed during the graph evaluation

How?

Package Descriptions

- Minimal version checking. Version numbers are only used to trigger different build instructions (add additional dependencies, or enable / disable features)
- If a package won't work with a random version of its dependency, then we prefer that package to just fail to build in that configuration. The most frequent developer interaction with a package is to update its version!
- Failed builds cannot be deployed!

How?

Target definitions

- List of final environments that ***must*** be built by the deployment
- Input the package parameters to drive the various combinations of artefacts (usually version numbers or compile options)
- These parameters can be nested and derived from to allow for minimal copy-pasting
- Are also created by executable code, allowing for loops or other more complicated definitions

```
var global_ctx = ctx.derive(get_default_versions());

var python3_ctx = global_ctx.derive({
    "python": new P("python", {"version": "3.7.3"})
});

var vfx2021_ctx = global_ctx.derive({
    'gcc' :     new P("gcc",     {"version": '9.3.1'}),
    'qt' :      new P("qt",      {"version": '5.15.0'}),
    'python' :  new P("python",  {"version": '3.7.3'}),
    'boost' :   new P("boost",   {"version": '1.73'}),
    'openexr' : new P("openexr", {"version": '3.0.0'})
});

var nuke_13_ilm = vfx2021.derive({
    'nuke':           new P("nuke",           {"version": "13.0v1"}),
    'qt':             new P("qt",             {"version": "5.12.x"}),
    'ilm_color_node': new P("ilm_color_node", {"version": "1.0.3"})
})

registry.register_target("global", global_ctx, ["bash", "coreutils"]);
registry.register_target("python3", python3_ctx, ["python", "bash"]);
registry.register_target("nuke13", nuke_13_ilm, ["ilm_nuke"]);
```

How?

Developer interaction

```
$> bb build gcc=9.1.0 openexr=2.3.0 python=3.7.3
Build successful! /artefacts/local/world/aa0e217
```

1. Downloads the latest copy of the deployment
2. Overrides the default versions for the given packages
3. Evaluates all the builder functions and checks hashes
4. Issues build commands in dependency order
5. Returns the path to the built environment

How?

Developer interaction

```
$> bb world /artefacts/local/world/aa0e217
$> gcc --version
gcc (GCC) 9.1.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$>
```

- Reads the “world” package provided and launches commands (defaults to bash)
- Has controlled interaction with the outside world

Technical Implementation

Technical Implementation

Details

- Most functionality is exposed via libbb, a self contained C library with very minimal external dependencies (only libm, libc, etc)
- The responsibility of entering environments is managed solely through the `bb world` command, meaning different platforms can have different implementations (eg, containers, LD_LIBRARY_PATH, Hyper-V, Hypervisor.framework)
- In the current Linux implementation, packages are brought together via overlayfs in their own process space via the unshare syscall

Technical Implementation

Why {javascript, lua, ?} ?!

- Build functions need to be evaluated often
- They have very few interdependencies and would be fastest if executed over multiple threads
- Most scripting languages are not implemented in a thread safe fashion (including Python)
- Would love to use Starlark (a python dialect used in Bazel), but implementations only exist in Go, Rust, and Java
- Support for other languages can be easily added in the future

Possible workflows

No blocked releases

Developers attempt to release their changes immediately, and rely on extensive automated unit testing / integration testing before deployment to catch bugs. Production has the ability to roll their entire show back to a known good point in time in the event of error.

Small batch testing

Developers can make temporary environments available to selected artists for testing, without worrying about impacting the rest of production, or having the test versions leak to other users.

Possible workflows

Easy off-site deployment

As the environments are utilising Linux containerization primitives, exporting from the build system as a docker container or similar would be possible, easing laptop or independent server deployment.

Extremely simple OS updating

As the build system only really relies on userspace of the Linux Kernel underneath, upgrading the operating system is very simple. Centos 7-8 migration would be seamless, and no chasing of packages would be required.

Possible workflows

Immutable and Reproducible

The internal hashing and read-only nature of the build system artefacts make for a great combination with asset management systems. Renders could store a dependency on a fixed moment in time of our software state.

Isolation and testability

Because every change to the deployment happens in isolation, it is possible for developers or IT support to try out massive changes without fear of breaking production, for example updating the compiler used to mitigate Spectre class vulnerabilities, or seeing if a database schema update would work out.

Possible workflows

Easier debugging

Developers have the ability to rebuild every package in their hierarchy in debug mode at the press of a button.

Easier troubleshooting

In DCC's with many plugins, it can be difficult to determine who is at fault for segfaults and crashes; bb makes it easy to build custom environments such as "Nuke with only 3pp plugins", "RV without GLSL nodes", "Maya with only this one plugin loaded".

Live Demo

Thank you

Questions?