

Isar Build Time Reduction

Felix Mößbauer

Siemens AG

About me



Felix Moessbauer

Senior key expert @Siemens FT
Embedded Linux consultant & developer



@fmoessbauer

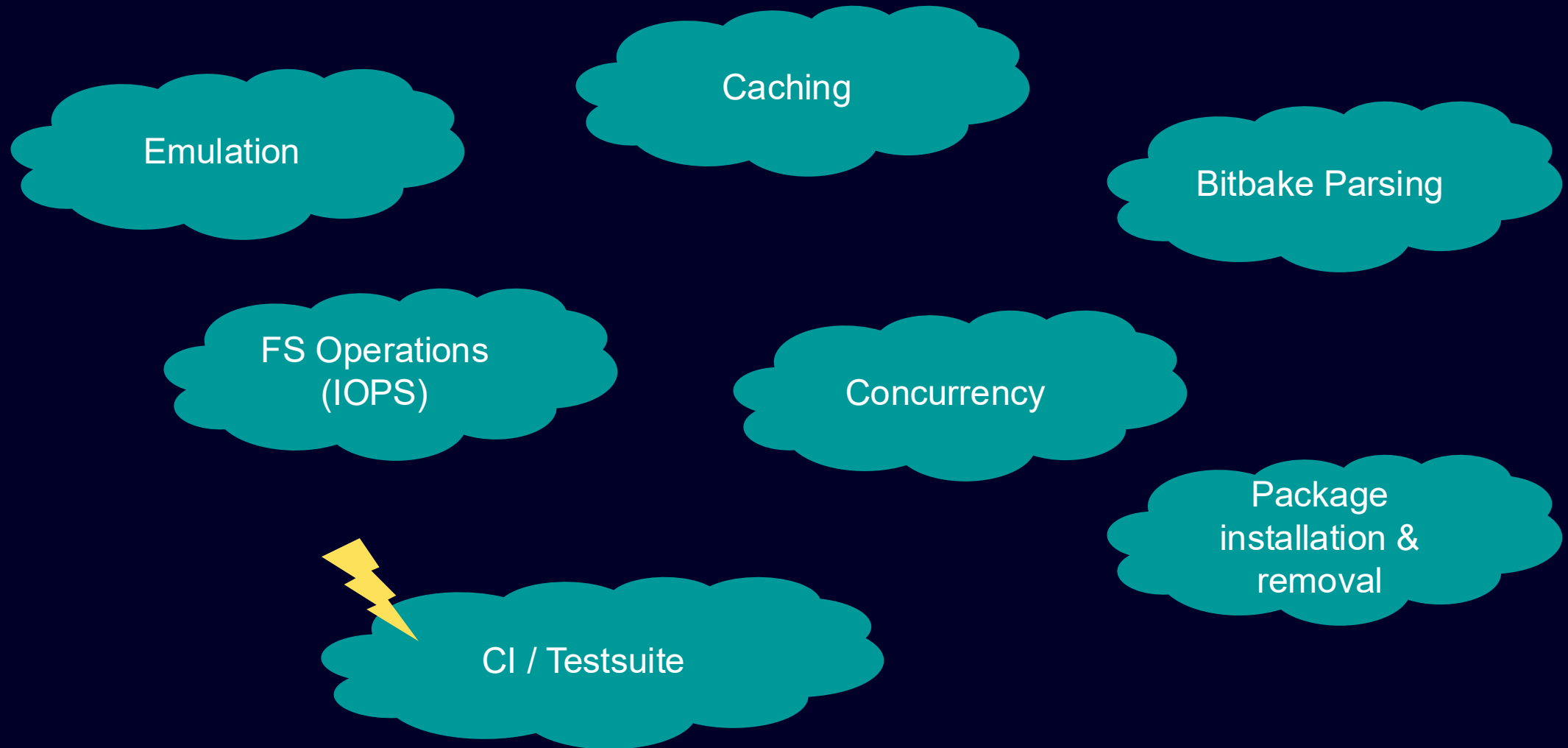


mastodon.social / @fmoessbauer



felix.moessbauer@siemens.com

Areas of improvement



Preparation: CI / testsuite

Speedup and improvements (1/2)

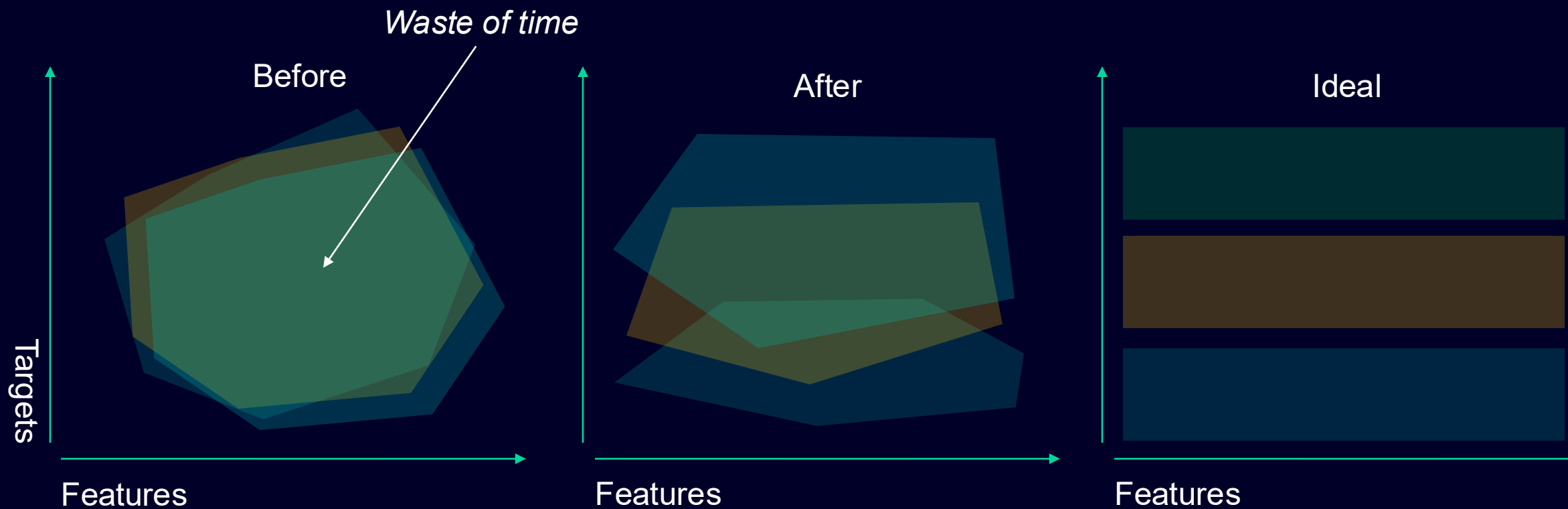
Problem

- For build-time reduction, a lot of places in isar need to be touched
- Ensure we don't break things while refactoring
- CI at project start, hardly usable (WAY too slow, inter-test dependencies, 5min+ parsing time per test)
- CI is hard to execute, comparing versions is tricky

Achievements

- Can be executed in kas-container now (testsuite/run-tests.sh <...>)
- Generate kas configs of single tests (for debugging): -p kasconfig=1
- Reduce inter-test dependencies, make them explicit
- Reduce parsing time (5min+ -> 10 sec)
- Allow sstate caching
- Reduce test overlap, cover more features

The isar CI in 2026



Emulation (1/5)

Building for non-native targets

- Operate from the outside
chroot `${rootfs}` find `/var/...` vs. find `${rootfs}/var/...`
- Locale generation and purge
 - Generation requires emulation, purging not
 - Configure dpkg to not extract things we anyways purge later
- apt operations -> use host apt
 - package downloading – emulates a whole http stack
 - do not install and remove a package
- Initrd generation
 - mkinitramfs emulates everything – thousands of calls to ldd, gzip, cpio, ...
 - dracut supports cross-mode -> most parts run natively

Emulation (2/5)

Using the host apt

- Switch sources.list between external and internal path ([file:///](#) entries are always resolved in calling env)
- Restore sources.list after external apt operations
- Install apt via ROOTFS_PACKAGES for sbuid chroots
- Apt runs outside chroot, dpkg inside

```
ROOTFS_HOST_APT_OPTS = "\n-o Dir=${ROOTFS_DIR} \n-o APT::Architecture=${ROOTFS_ARCH} \n-o DPkg::Chroot-Directory=${ROOTFS_DIR} \n"
```

```
HOST_APT_CMD = "\nenv APT_CONFIG=${ROOTFS_HOST_APT_CONFIG} \n/usr/bin/apt-get ${ROOTFS_HOST_APT_OPTS} \n"
```

Emulation (3/5)

Dracut cross

- Dracut supports initrd assembly "from outside"
 - Modules need to support this
 - Fine grained control over which tool runs outside and inside the chroot possible
 - Avoids costly emulation of ldd, cpio and compression
- Speedup (arm64): 10min -> 2min

```
ROOTFS_INITRAMFS_GENERATOR_CMDLINE: \  
  append:dracut-cross = " --sysroot /mnt/rootfs" \  
DRACUT_ENV = "DRACUT_ARCH=${QEMU_ARCH}" \  
DRACUT_ENV:append:dracut-cross = " \  
  DRACUT_INSTALL=/usr/lib/dracut/dracut-install \  
  DRACUT_LDD=/usr/libexec/dracut-cross-ldd \  
" \  
 \  
export ${DRACUT_ENV} \  
chroot ${DRACUT_ROOTFS} \  
  ${ROOTFS_INITRAMFS_GENERATOR_CMDLINE}
```

Dracut support will be available in isar-cip-core soon!

Emulation (4/5)

Preparing modules for dracut cross

```
--- a/meta-isar/recipes-initramfs/dracut-example-lighttpd/files/install.sh
+++ b/meta-isar/recipes-initramfs/dracut-example-lighttpd/files/install.sh
@@ -1,5 +1,9 @@
-inst_multiple -o /usr/lib/lighttpd/*.so
-inst_multiple -o /usr/share/lighttpd/*
+# Copy optional lighttpd modules and assets from the target sysroot.
+for source in "${dracutsysrootdir}"/usr/lib/*/lighttpd/*.so \
+             "${dracutsysrootdir}"/usr/share/lighttpd/*; do
+    [[ -e "$source" ]] || continue
+    inst_simple "$source" "${source#"${dracutsysrootdir}"}"
+done
```

Wildcard resolves in
wrong environment

Emulation (5/5)

Preparing modules for dracut cross

```
-/usr/bin/install -m 0644 /usr/share/lighttpd/index.html "$initdir/var/www/html/index.html"
+inst_simple "${dracutsysrootdir}"/usr/share/lighttpd/index.html /var/www/html/index.html
  touch "$moddir"/error.log
-/usr/bin/install -m 0644 "$moddir"/error.log "$initdir/var/log/lighttpd/error.log"
+inst_simple "$moddir"/error.log /var/log/lighttpd/error.log
```

```
install() {
    for executable in ${DRACUT_REQUIRED_BINARIES}; do
-        if exec_path=$(command -v $executable 2>/dev/null); then
+        if exec_path=$(find_binary "$executable"); then
            inst_binary "$exec_path"
        else
            echo "(ERROR): Unable to copy $executable" >&2
        fi
    done
}
```

Use dracut tooling,
not shell tooling

Filesystem operations & parsing

IO is costly

Filesystem

- dpkg
 - do not extract data that is removed later on
- internal compression
 - use zstd wherever possible (no gzip or xz)
 - do not double-compress (-> sstate)
- minimize rootfs artifacts
 - cleanup prior to tar'ing
 - do not include apt cache

Parsing

- multiconfig is super-expensive
 - only generate multiconfig entries for targets that are actually built
 - memorize process calls (immediate eval)
(`R := <exec>` instead of `R = <exec>`)
- rewrite expensive inline python expressions (e.g. `generate_distro_sources`)

Many tasks are IO bound -> a more powerful machine does not help!

Concurrency & SState caching

race conditions, implicit dependencies, redundant work, correctness issues

Observations

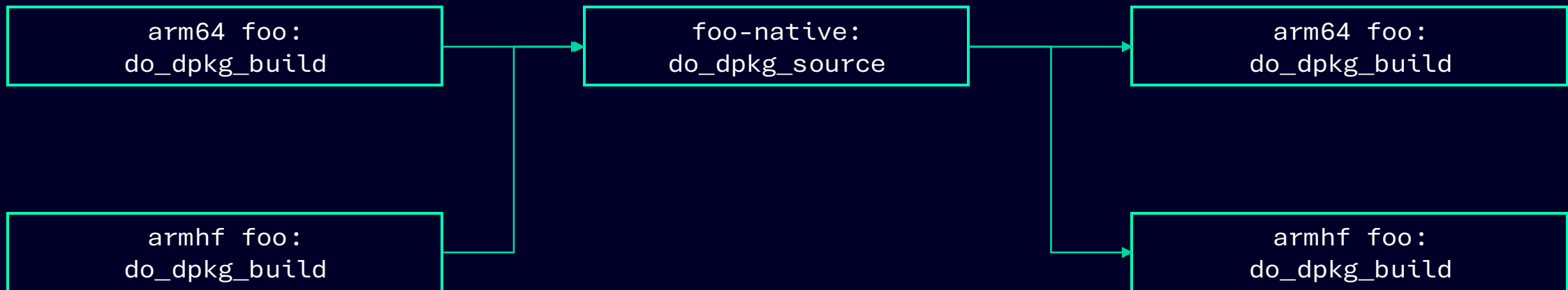
- caching improvements surface sleeping bugs!
 - 10+ commits fixing these
- better caching -> task order changes in unexpected ways
- groups of tasks behind an sstate cached task are executed concurrently (.e.g do_deploy_deb) -> race conditions
- caching across multiconfig targets -> tons of race conditions
- redundant work due to non-deterministic cache inputs -> sometimes incorrect data in cache
- missed tasks due to forgotten dependencies (e.g. deploy of source package into isar-apt)

Concurrency & SState caching

Source packages, binary packages and multiarch ($\${PN} \neq \${BPN}$)

Observations

- Source packages are architecture independent
- Binary packages are arch dep, except for arch: all
- A source package generates 1:n binary packages
- Modelled via SRCPKG_PROVIDER = " $\${BPN}$ -native"



Concurrency & SState Caching

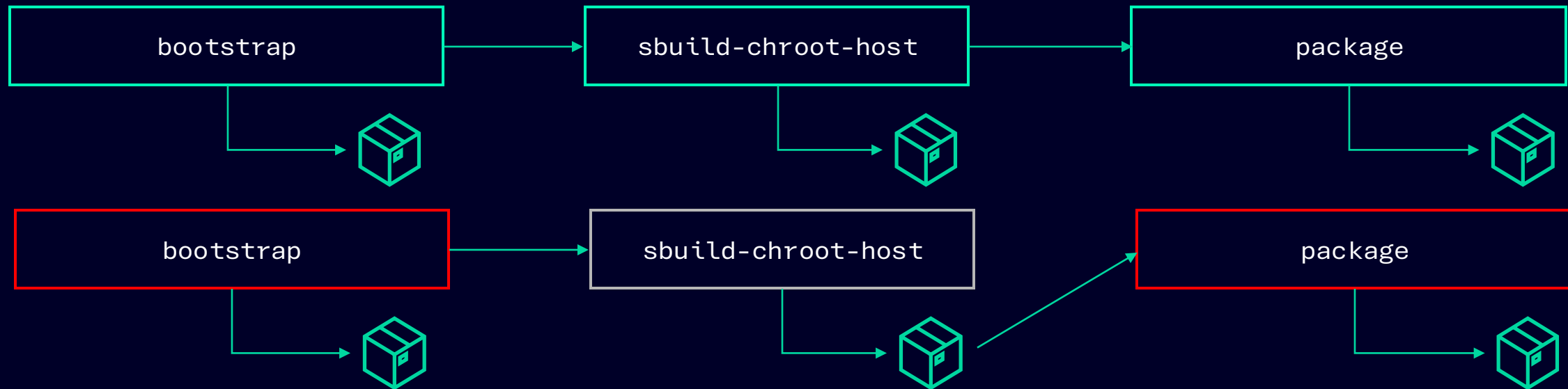
Hash equivalence

How does it work

- taskhash: Inputs of a task (variables, code, ...)
- outhash: hash of generated artifacts (sstate)
- unihash: unified identifier assigned to a task to determine cache reuse.

Unihash

- Initially equal to taskhash
- After running, bitbake reports taskhash and outhash to hash-equiv server



[1] Reproducible Builds and Hash Equivalence in the Yocto Project - Joshua Watt, Garmin <https://www.youtube.com/watch?v=zXEdqGS62Wc>

Concurrency & SState caching

Sources of non-reproducibility

- isar-apt
- Timestamps
- Caches in rootfs
- Logfiles in rootfs
- Pycache (.pyc) on pre trixie images (technically not a cache)
- Tooling output (compression, ...)
- External changes (e.g. deb.d.o instead of snapshot.d.o)
- Absolute paths in bitbake variables (e.g. GNUPGHOME),
`d.setVar( 'X' , d.getVar( 'Y' ) )`

Analyze with diffoscope

Analyze with isar-sstate lint, delta

For hash-equivalence, all sstate artifacts must be reproducible, not just the final image!

Challenges

Observations

- Isar areas I'm not familiar with (base-apt)
- Bugs shadowed by other bugs (docs are not fully removed, but healed by later cleanup)
- Dead code
- Missing test coverage

General design issues

- SDK: including isar-apt is non-deterministic because isar-apt itself is not bound to the rootfs
- Multiconfig builds with two machines having common distro and arch
- No bitbake update possible anymore: parts of bitbake run inside chroot, hence need to support debian buster

Summary & Outlook

Speedup: Depending on use-case, 3x build speedup on repeated builds is possible

Functional: No need to have apt in the final image
-> reduced image size, simpler license clearing

Siemens AG

Felix Moessbauer

felix.moessbauer@siemens.com

Future ideas

- Provide public sstate mirror (similar to Yocto)
- Bitbake update (image from outside the chroot)
- Optimize feature coverage in CI (test less, gain more)