CONTINUOUS INTEGRATION AND TESTING OF A YOCTO PROJECT BASED AUTOMOTIVE HEAD UNIT

MARIO DOMENECH GOULART
MIKKO RAPELI

Embedded Linux Conference Europe 2016
ABOUT BMW CAR IT GMBH

– Founded in 2001 as a wholly owned subsidiary of the BMW AG
– Strengthen BMW's software competence
  – View vehicles as software systems
  – Develop innovative software for future BMW Group vehicles
  – Prototype solutions for early and reliable project decisions
– Participate in several open-source communities and research projects
PROJECT SETUP

– Development of a head unit for BMW cars
  – A connected multimedia computer with navigation and telephony
– Several companies, physically distributed
– Hundreds of developers, on various levels
– Complex infrastructure
– Technical and political obstacles to set up technical solutions
CI SYSTEM REQUIREMENTS

– Provide fast feedback for developers, integrators, project organization
– Automatic multi-stage CI
– Software components change-verification in an SDK environment
  – Build components
  – Execute unit tests
– Software integration change-verification in the system build
  – Build the full system, for all targets, all images
  – Quality assurance checks after build
  – Build Acceptance Testing (BAT) on real target environments (hardware, SDK)
QUICK OVERVIEW OF YOCTO PROJECT

– Linux-based cross-compilation framework
– Set of metadata and a task scheduler which, combined, can be used to build software
  – Metadata
    – Configuration files. Examples:
      – Machine configuration (target platform)
      – Target Linux distribution configuration
  – Recipes
    – Specification of tasks on how to build software (fetch, configure, compile, package etc.)
    – References (e.g., git URL and commit id) the actual source code of the component it describes
  – Tasks can be implemented in Python or Shell scripts
  – Maintained in separate meta repositories (e.g., git repository)
QUICK OVERVIEW OF YOCTO PROJECT (CONTINUED)

- Task scheduler: BitBake
  - Inputs: metadata
  - Outputs (typical use): packages, images, toolchains, SDKs etc.
- Sysroots
  - Global staging area for builds
  - Where build dependencies are installed during build
  - Shared among all build tasks
- Caching
  - Shared State cache (sstate cache)
    - Cache of processed BitBake tasks
  - Download cache
    - Cache of source code (git, subversion, tarballs etc.) downloaded by BitBake
YOCTO PROJECT: NEAT FEATURES AND CHARACTERISTICS

- Very flexible
  - Fine-grained control on artifacts
  - Compile-time configuration
- Extensible
  - It's easy to add your own metadata or extend existing ones by adding layers
- License tracking
  - You can specify what licenses your product cannot ship
- Support
  - Commercial support
  - Community support
- QA checks
  - Help to catch problems earlier
SOURCE CODE MANAGEMENT
SOFTWARE COMPONENTS

– Public open source (git, tarballs, etc.)
– Internal projects (git)
– Binary software deliveries from suppliers (subversion)
SYSTEM COMPONENTS

- Yocto Project (git)
- Open source meta layers (git)
- Proprietary meta layers (git)
- All system components are git repositories assembled as git submodules in a single base git repository
- Each commit in the base repository represents the full state of all the git repositories
- Testing changes that affect multiple submodules is easy (e.g., Yocto Project updates)
- Drawbacks
  - Confusing for developers new to git
  - Adding and removing submodules cannot be easily tested in CI
  - Not nicely integrated to Gerrit, Gitweb or git GUI tools
- Alternatives
  - Repo
  - Custom scripts that save state somewhere
GERRIT

– Hosts git repositories for software and system components
– Topics to group commits that affect multiple repositories
– Custom tool to check out topics into a working tree (python, gerrit API’s)
– CI jobs can verify all changes with the same topic
– Positive aspect: for experienced developers this setup works well (local feature branch == topic)

– Drawbacks
  – Inexperienced developers make mistakes
  – Mixing unrelated changes in a single git repository, under the same topic
  – Trying to merge commits that are not part of the same branch
  – Gerrit UI is confusing
  – Corporate IT hosted Gerrit is not up-to-date with upstream Gerrit

– Alternatives
  – Patchwork/e-mail
    – E-mail is a nightmare in corporate environments (Outlook, MS Exchange, HTML, Windows users etc.)
  – Github, Gitlab (we haven't tried them)
SOURCE CODE CHANGE INTEGRATION

– In the software component we apply changes with Gerrit (apply and merge)
– In the system integration we create pull requests that involve multiple git repositories
  – e.g., a Gerrit topic that contains changes in multiple repositories
  – Pull requests are called Integration Requests (IR) in our process
  – Integration requests can only be issued after a positive peer review in Gerrit and successful verification build in CI
– CI system merges and tests the merged changes before release
OVERVIEW OF THE CI PIPELINE
SOFTWARE COMPONENT DEVELOPMENT

– Software component developers work with the SDK
– Push changes to Gerrit code review
– Gerrit triggers a verification build with the SDK (includes unit tests)
– In case of successful verification, changes can be merged automatically or manually
SYSTEM INTEGRATION

- Two types of integration requests
  - Automatically/manually submitted from a component repository
    - The git commit hash in a BitBake recipe is changed
  - System integration Gerrit topic affecting multiple git repositories
MULTI-STAGE CI

- SDK verification
- System build
- Merge verification before release
SDK VERIFICATION FOR SW COMPONENTS

- Developer
  - Code review result
  - Git changes
  - SW component git
    - Git changes
      - Build and unit test
        - SDK verification
          - SDK from latest release
            - SW component releases to system integration
              - Developer
                - Git changes
                  - SW component git
SYSTEM CHANGE VERIFICATION

- **Integrator**
  - Code review result
  - Integration Request (IR, pull request for multiple git trees)
  - SW component releases to system integration

- **Base repository and meta layers**
  - Git changes
  - Verification result

- **System build and BAT tests**
  - Git changes
  - Caches from latest release: sstate, download

- **Integrator**
  - Git changes
SYSTEM RELEASES

Integration Request (IR, pull request for multiple git trees)

Release managers

Yes/no/not yet

Change Control Board

IR for merge

Base repository and meta layers

Verification result

Releases:
- Tagged git tree
- Release artifacts, images
- SDK, caches etc

System build and BAT tests

Further testing phases...

Yes/no/not yet
AUTOMATIC RELEASE MANAGEMENT

- Integration requests are applied and tested in a full system build
- Change Control Board can control which integration requests get merged
- A set of integration requests are collected and pushed out as a release
- New releases can be created manually or based on timer
CI INFRASTRUCTURE
- Gerrit, git and subversion servers
- Jenkins servers (several masters and even more slaves)
  - Predominantly virtual machines
  - Build slaves (SDK and BitBake builds)
  - SDK build slaves: 45 (8 CPUs, 20GB of RAM)
  - BitBake build slaves: 36 (16 CPUs, 64GB of RAM)
    - Two bare metal machines (no virtualization): 40 CPUs, 128GB of RAM
    - One daily build from scratch (without sstate cache)
- File and cache servers
- Database server
- Cluster of virtual machines
- Bug and issue tracking servers
- Test farm with special hardware, including target hardware devices
  - Jenkins masters have test jobs which are triggered by build jobs
  - Custom Python-based test farm framework uses RabbitMQ to trigger test executions on the test farm
- Test farm has 16 SDK, 20 virtual targets and 12 real target executors
- Besides the test farm we also have automated tests for the build artifacts
  - Test as much as possible without the target platforms
TEST FARM STATISTICS (1)
### TEST FARM STATISTICS (3)

![Graph showing average test suite duration](image)

#### Average test suite duration (not considering filters and exit_code 0|1)

<table>
<thead>
<tr>
<th>Test Run</th>
<th>Max</th>
<th>Avg</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_run</td>
<td>14.5 min</td>
<td>13.8 min</td>
<td>12.8 min</td>
</tr>
<tr>
<td>test_run</td>
<td>37.3 min</td>
<td>26.9 min</td>
<td>29.6 min</td>
</tr>
<tr>
<td>test_run</td>
<td>22.7 min</td>
<td>14.8 min</td>
<td>13.6 min</td>
</tr>
<tr>
<td>test_run</td>
<td>21.1 min</td>
<td>20.4 min</td>
<td>19.8 min</td>
</tr>
<tr>
<td>test_run</td>
<td>18.3 min</td>
<td>8.3 min</td>
<td>9.1 min</td>
</tr>
<tr>
<td>test_run</td>
<td>27.4 min</td>
<td>13.1 min</td>
<td>12.3 min</td>
</tr>
<tr>
<td>test_run</td>
<td>12.6 min</td>
<td>5.0 min</td>
<td>2.3 min</td>
</tr>
</tbody>
</table>
LESSONS LEARNED

– Keep it simple
– Solid foundations
  – Use real distributed system technologies, not hacks on top of Jenkins and regular file transfer tools
– Corporate networks are sometimes less reliable than Internet services
– Automate everything (ansible, puppet etc.)
– Virtualization is not an ideal solution when it comes to performance
LESSONS LEARNED (CONTINUED)

– Positive aspects
  – It works, although sometimes administering the system is painful
  – It fulfils the requirements of the project as a CI system

– Negative aspects
  – Jenkins is not a distributed system
  – Not everything is automated
  – Some changes in the CI infrastructure cannot be tested by the CI system
BUILDS
SOFTWARE COMPONENT BUILDS

- Use the SDK provided by BitBake builds
- SDK can be extended with packages, automatically in CI jobs, or manually by users
- ccache is used to make builds faster
SYSTEM BUILD

- Runs inside a LXC container with Ubuntu 14.04
- The container
  - Provides build isolation
  - Can be constructed during build (e.g., container changes can be tested in the CI)
  - Mitigates host contamination
    - Prevents system components to leak into the build environment
- The influence of the host system in the build is at least reproducible
- Container changes can be deployed faster than changes in the infrastructure
- Developers are free to use any Linux distro they want and still use the container for building
SYSTEM BUILD - IMPLEMENTATION

- Wrapper shell script around BitBake, for each target machine
  - In CI builds, synchronizes the sstate cache from the previous release before calling BitBake
  - In CI builds, used a mounted NFS share for the download cache
  - Developers are out of luck with regard to caches, due to network setup complexity
- Lesson learned
  - Bash and set -eux -o pipefail, at least
  - Cleanup in trap commands
SYSTEM BUILD – META LAYERS

– Each meta layer is a single git repository with a single owner (a team)
– The owner has +2 review rights for its git repository
  – A change gets approved if it gets a +2 from review and a +1 from the verification build
– More than 60 meta layers
– More than 2800 recipes
– More than 400 bbappends
SYSTEM BUILD – BITBAKE CONFIGURATION

– template file for `local.conf`
– `sed` magic for environment-dependent configuration options (e.g., mirrors and network usage metrics)
– custom script for setting BitBake parallelization options based on the number of CPU cores and RAM (details later)
“all” is a special BitBake recipe that specifies everything to build

- Multiple images for the target hardware (“boot modes”)
  - Image artifacts include flashing and testing tools
  - Images are tarballs, not filesystem images (flashing creates filesystems)
  - Building an image is a serial operation (cannot be parallelized)
  - Multiple images can be built in parallel, but not the installation of packages in a single image
  - Images share a lot of content, but we don't have a way to reuse the common parts
  - The target images have big data blobs that we manage with git annex (plugged into BitBake)
  - Image tarballs are compressed with pigz for parallel compression (using multiple CPUs)
  - Support for filesystem extended attributes is needed in the future
SYSTEM BUILD - SDK

- Custom SDK instead of Yocto Project upstream
  - In the SDK we mix target and nativesdk packages, in a way that it is transparent for users
- Motivation
  - Developers struggled with the cross toolchain and cross environment setup
    - Mistakes in the development of components’ build system (CMake)
    - Complexity of the cross-compilation environment shifted from developers to the integration team
- SDK content decoupled from images
- Custom namespace tooling instead of plain chroot (execution environment for the SDK, without root access)
  - Transparent cross-compilation in the SDK, using gcc, make, autotools, cmake and other tools from $PATH
- From users perspective, it looks like a lightweight chroot
Automated CI tests for everything that we add to the SDK

- Even trivial tests find bugs

- It would be possible to run upstream Yocto Project's SDK tests in our SDK (some minor fixes are needed)

- Users and CI jobs can extend the SDK with packages

Qt Creator IDE with custom plugin to ease the development using the SDK

Our SDK approach and tests have not yet been upstreamed

- Planned for one of the next iterations

The SDK contains tools and tests for the CI automated tests
SYSTEM BUILD – PACKAGE ARCHIVE

– Format: ipk
– Package archive with additional tools, debug symbols, development packages etc.
– Due to the complexity of corporate networks, we could not set up a single package repository server
– We distribute packages to a number of mirrors in different networks (even using different protocols)
– Some debugging tools are only available in the package repository
– We don't support incremental updates of SDK and images using the package repository yet
  – Due to the complexity of the network setup, we don't have a PR server
  – We bump PRs manually
– We plan to reuse the PR server database files
SYSTEM BUILD - DIFFICULTIES WITH YOCTO PROJECT

– Writing proper BitBake recipes is a form of art - only a few people know how to do this correctly
  – BitBake is too flexible - too much freedom
– The shared sysroot approach in the context of parallel recipe processing causes build race conditions
  – Some software enable/disable features based on the state of sysroots
  – The state of sysroots vary as build tasks are executed
  – Undeclared build dependencies often go unnoticed
– Developers add features to their software, but forget to specify dependencies in recipes
  – Sometimes packages build fine on populated sysroots, but break due to missing dependencies specification when built from scratch
  – Developers and CI build images, instead of changed recipes with an empty sysroot
  – Sstate cache hides problems until something triggers a rebuild
– Floating build dependencies
  – Features are implicitly enabled/disabled based on the state of sysroot
  – May cause build or test failures
– In our case, BitBake builds are not reproducible
– Packaging of language extensions (e.g., Java's maven, JavaScript's npm) is problematic
– Using specific package managers just hides the problem and lead to not reproducible builds
– Developers may call package managers like maven from their build scripts while generating code
  – Downloading modules from the Internet may fail
  – No guarantees with regard to integrity of downloaded modules
  – No sum checking and no caching on the BitBake side
  – May break packaging
  – No license tracking
– BitBake rebuilds dependents even when it is not strictly required
  – API/ABI compatibility is preserved
  – Leads to long build times
SYSTEM BUILD - NUMBERS

- For “all” (per target machine)
- More than 22K BitBake tasks
- More than 8K packages generated (~6.4GB)
- One SDK
  - ~600MB
  - ~1100 packages
- Nine images (numbers on the biggest):
  - ~510MB
  - ~845 packages
SYSTEM BUILD - PROFILE

- Build times may range from 20 minutes to 5 hours
- Build performance can be hard to optimize
  - Many variables to tweak
  - Different build characteristics, depending on what has to be compiled (BitBake caches)
- Some heavy-weight components
  - Big C++ components
  - Some of the big ones are affected by dependencies that change frequently, so they have to be rebuilt
- Several build steps cannot effectively utilize multiple CPUs
  - Some tasks like do_rootfs (image creation)
  - Run queue preparation
- buildstats data can be useful to understand builds
SYSTEM BUILD - POSTPROCESSING

– Check the presence of expected files
– Sstate cache preparation after releases
– Publishing of artifacts (packages, images, SDK, logs etc.)
– After a release, a new SDK is deployed into the system
DETERMINE BOTTLENECKS

- System resources
  - CPU
  - Memory
  - Disk I/O
  - Network I/O
- Require system monitoring tools
  - Performance co-pilot (pcp)
  - htop
  - buildstats
  - syslog
  - Grafana
DOWNLOAD CACHE

- Alleviates the load on some slower paths in the company's network
- A special BitBake job (-c fetchall) populates the cache into a NFS share which are mounted by the build slaves
  - Does not fully validate the downloads after bitbake -c fetchall
  - Corrupted downloads lead to build failures
- Ideally, we would like to be able to run offline builds (no network)
BITBAKE PARALLELIZATION SETTINGS

- BB_NUMBER__THREADS, PARALLEL_MAKE
- The default parallelization options set by BitBake don't work for build profile
  - Compilation of a single C++ file can consume gigabytes of physical RAM
  - Example: machine with 16 CPU cores (PARALLEL_MAKE=16, BB_NUMBER_THREADS=16)
    - Worst case: 256 compilation tasks running at the same time
  - We observed system load above 100
  - Some builds run out of RAM, which leads to heavy swapping or OOM killer (breaks builds)
- Lesson learned
  - Measure and set resource limits for BitBake tasks (cgroups)
  - Ideally, the BitBake scheduler should take into account the system load when scheduling
  - Should not spawn tasks when load and memory usage reach some limit
OPTIMAL PARALLELIZATION IS HARD TO GET

- In cases of lots of caching, high parallelization is desired
- In cases of low caching, high parallelization may lead to system trashing due to high resource usage
- We use a custom script to set up parallelization options which takes number of CPU cores and RAM into account to set the parallelization options
mem = get_mem_total()
cpus = get_number_cpus()
mem_cpus = (mem * 1.0) / cpus

if ncpus == 1:
    BB_NUMBER_THREADS, PARALLEL_MAKE = (1, 1)
elif mem_cpus > 8:
    BB_NUMBER_THREADS, PARALLEL_MAKE = (cpus, make_j(cpus))
elif mem_cpus >= 4:
    BB_NUMBER_THREADS, PARALLEL_MAKE = (cpus, make_j(divide_cpus(cpus, 2)))
elif mem_cpus >= 2:
    BB_NUMBER_THREADS = divide_cpus(cpus, 2)
    PARALLEL_MAKE = make_j(divide_cpus(cpus, 2))
else:
    BB_NUMBER_THREADS = divide_cpus(cpus, 2)
    PARALLEL_MAKE = make_j(divide_cpus(cpus, 4))
BUILD SLAVE TUNING

- Avoid "disk" I/O
  - Keep data on memory for as long as possible (Linux memory manager settings - sysctl)
    - `vm.dirty_background_bytes = 0`
    - `vm.dirty_background_ratio = 90`
    - `vm.dirty_expire_centisecs = 4320000`
    - `vm.dirtytime_expire_seconds = 432000`
    - `vm.dirty_bytes = 0`
    - `vm.dirty_ratio = 60`
    - `vm.dirty_writeback_centisecs = 0`
- Avoid swapping
  - Lots of RAM help (up to certain point)
    - Increasing RAM from 64GB to 128GB on a machine with 40 CPU cores didn’t improve build times
- More aggressive parallelization options lead to system trashing, thus slower builds
- Solution: experiment; profile the build and tune resources and parallelization options
QUALITY ASSURANCE AND SECURITY
STATIC CODE ANALYSIS USING CODE SONAR

- Finds CERT programming errors like memory leaks, buffer overflows and race conditions
- Similar to Coverity
- All the BitBake recipes are recompiled using Code Sonar's compiler wrapper
- Slow: takes roughly five days
- Automated, but not directly connected to the CI workflow
OPEN SOURCE LICENSE COMPLIANCE

- We use the license information provided by BitBake recipes
- Additionally, we use Black Duck’s Protex to analyse source code for cases of license violation
- Automated, but not directly connected to the CI workflow
SECURITY VULNERABILITY ANALYSIS

– We need to know which CVEs affect our products
  – Tooling provided by Yocto Project patches
  – Black Duck also supports this, but we are not using it yet
CONCLUSIONS
ON YOCTO PROJECT

– Community support on mailing lists, IRC and bug tracker is good
– Documentation is good, but the system is complex
– Yocto Project's core meta layers are our reference in terms of quality
– It's difficult to achieve the same level of quality as Yocto Project's in our meta layers
– Some fundamental BitBake design decisions cause us some problems
  – Shared sysroots lead to build race conditions and dependency issues
  – Huge amount of global, mutable variables
– No reproducible builds (in our case), even with the use of standard build environment (container)
  – We are working on making them reproducible and intend to have this feature by the time we ship the product
LESSONS LEARNED ON THE DESIGN OF OUR CI SYSTEM

- CI systems can be used to automate any task of the development process
- CI software builds find bugs
- CI tests, even if trivial, also find bugs
- Cultural change: some developers and project partners appreciate the feedback of the CI system
- Cultural resistance: some project partners and developers don't
- Quality of service in corporate network makes the implementation of CI systems difficult, reliability suffers
- Reliability of the system depends on the reliability of the parts (hypothetical example):
  - Source code servers: 95% availability
  - Build reliability: 90% and then developers changes on top
  - Tests: 90% reliability

  => $0.95 \times 0.90 \times 0.90 = 76.9\%$ overall reliability
Mario Domenech Goulart
mario.goulart@bmw-carit.de

Mikko Rapeli
mikko.rapeli@bmw-carit.de