Tools and toolchain SIG

1. Summary

Tools and toolchains are an important part of building any operating system. They have a big impact on performance, compatibility and developer friendliness.

It is therefore important for OpenHarmony to make the right decisions about tools and toolchains, making a SIG necessary.

2. Motivation

2.1 Goals

1. Determine the best tool and toolchain options currently available (see 3. for examples)
2. Optimize the chosen components for OpenHarmony – set reasonable defaults with OpenHarmony triplets, make sure OpenHarmony is a visible target in IDEs, etc.
3. Push modifications to toolchain projects like LLVM and gcc upstream – any local patches to toolchain components should be temporary. This includes helping other teams like Harmony (non-Open) upstream their changes, if any/if needed.
4. Keep monitoring the Open Source world for new versions (the toolchain should track upstream releases – they usually come with better performance and compliance with newer language standards) and entirely new projects (including, but not limited to, Huawei’s OpenArk compilers) that could improve OpenHarmony tools and toolchains, update OpenHarmony tools and toolchains in a timely manner
5. Provide expertise/assistance to developers running into toolchain problems (compiler bugs etc.)
6. Make sure people trying to build OpenHarmony and/or develop OpenHarmony apps can easily install the needed toolchain components regardless of their work environment
7. Identify and adapt debugging, profiling, and code quality tools and sanitizers that can be useful for the OpenHarmony ecosystem. If a new tool needs to be developed, try to develop it inside an upstream project to bring in external developers/maintainers.
8. Decide on components and architecture of the OpenHarmony SDK for hardware makers and application/game developers
9. Avoid host tool contamination – help make sure the OpenHarmony toolchain is used instead of potentially equivalent host tools that may come with a different set of bugs

2.2 Non-Goals

1. Writing new toolchain components from scratch
2. Forking toolchain components (if they can’t be avoided altogether, any forks with extra OpenHarmony support should be temporary until patches are accepted upstream)

3. Proposal

The Tools and toolchain SIG is launched, and starts looking into the following questions (more will be added later as OpenHarmony progresses):

1. gcc or clang?
2. libgcc or compiler-rt?
3. libstdc++ or libc++?
4. Do we want to include (optional) support for additional programming languages, such as Java, C#, Go, Rust, Swift? If so, which languages and which implementations thereof? Which runtimes (if any) go into the core OS, which are provided as optional components inside the SDK?
5. libc: Currently Linux and LiteOS-A use Musl, Zephyr uses a newlib fork, FreeRTOS is typically used with newlib. There are good reasons to use different libcs here (newlib doesn’t support dynamic linking, which is very important for higher end systems, while musl doesn’t support MMU-less systems), but there may be some things we can do to improve compatibility between OpenHarmony on top of different kernels, and some ways to optimize both worlds while reducing maintenance work (e.g. unify the implementations of some functions in both libcs, or even bring in more optimized versions of some routines from other libc implementations)
6. Which debugging tools, profilers, code quality tools, sanitizers etc. do we need/want on OpenHarmony?
7. What IDE or IDEs, if any, do we want to modify for improved OpenHarmony support? How can we best support IDE users and traditional tool users alike?
8. Do we assemble the toolchain (not its components; that is beyond the scope) from scratch or do we base our work on an existing project (e.g. OpenEmbedded/Yocto, Buildroot, a Linux distribution’s packaging, ...)

Once decisions are made, the SIG builds and maintains the tools and toolchains used in OpenHarmony and the SDK (tools, library headers, etc. and optional IDE(s) as separate modules).

The toolchain is provided in binary form and in source form with a simple build script that includes all configurations etc. used for the official build.

If a tool bug is reported to the SIG, it responds by fixing the bug or passing the bug report on to the corresponding upstream project with all needed information (possibly adding details like a fully reduced test case for a compiler bug to the original report), and by providing a workaround if the problem can’t be fixed properly in a timely manner.

Patches for bug fixes and optimizations should always go upstream in a timely manner. Local patches to upstream components should be temporary at most. This minimizes overhead and maximizes reuse.

3.1 User Stories

1 Typical app/game developer

A developer wants to build an app that runs on OpenHarmony. Thanks to the work of the SIG, he can install the needed SDK/crosscompiler easily (ideally it is available in his Linux distribution’s packaging), and because the SIG has been keeping the toolchain up to date, he can use current versions of the standards (such as C17, C++20) instead of being restricted to C11 and C++14.
Debugging/profiling tools provided by the SIG help detect and fix bugs in the app being developed.

2 Advanced app/game developer

A developer wants to build an app/game that runs on OpenHarmony. This developer has been developing code for various systems for years if not decades. He is used to his usual work environment (e.g. vim, emacs) and doesn’t want to learn/use a new IDE instead of the tools he is already productive with. Thanks to the work of the SIG, the components are not tied to any particular IDE and can be installed individually, and can fit the experienced developer’s workflow instead of forcing him to start over.

3 OpenHarmony developer

A developer working on OpenHarmony itself benefits from the same effects as the App developers, and more: While developing a new feature stressing the compiler, he runs into a compiler bug (e.g. “Internal Compiler Error”). He can turn to the SIG and get people familiar with the toolchains to look at the problem and identify a fix or workaround, and get help obtaining the information a good upstream bug report will want (such as a fully reduced test case).
Debugging/profiling tools provided by the SIG help detect and fix bugs in OpenHarmony.

3.2 Notes/Constraints/Caveats

N/A

3.3 Risks and Mitigations

Risk: The SIG makes a decision that later turns out to have been a bad choice (e.g. switching to a compiler that is later abandoned upstream – highly unlikely, but possible)

Mitigation: The SIG’s decision is revised and OpenHarmony migrates to a better tool.

4. Design Details

When investigating a tool, the Tools and toolchain SIG has to keep the following criteria in mind:

1. Tool must be available under a compatible Open Source license
2. Tool should have a clear future
3. Tool should deliver the best performance among comparable tools (or be significantly better in code quality and/or development speed, to the point that we can be reasonably sure it will match/outperform comparable tools in the not too distant future)
4. Tool should not be limited to a particular host system
5. Where possible, tools that can be shared across different target systems (hardware architecture as well as kernel choice) should be preferred over tools that introduce unnecessary differences. While some toolchain components may need different builds for Linux based OpenHarmony, Zephyr based OpenHarmony, LiteOS based OpenHarmony and FreeRTOS based OpenHarmony, they should be built from the same source.
6. While it likely makes sense to provide an easy to use IDE, a developer's choice should not be limited to that IDE. Developers using an IDE should be supported just as well as developers using more traditional tools like text mode editors and commands.
7. Where at all possible, situations like "Board X needs an old kernel that can be built only with an ancient toolchain, so we use a separate kernel compiler and userland compiler" should be avoided – preferably by keeping kernel support for relevant devices up to date, if necessary by backporting patches to support newer toolchains to kernels that are still relevant.
8. Code reuse should be maximized, and porting/rework efforts should be minimized.

4.1 Graduation Criteria

N/A

4.2 Upgrade/Downgrade Strategy

The SIG should upgrade toolchains in a timely manner to keep state of the art performance and language support.

Toolchain upgrade cadence should generally follow upstream toolchains as well as major OpenHarmony releases, with the possibility to skip a component release if they happen too frequently (e.g. LLVM major releases usually follow a 6 month cadence, major gcc releases are usually at least a year apart).

Downgrades should never be necessary.

5. Implementation History

N/A

6. Drawbacks

N/A

7. Alternatives

The alternative is not making our own toolchain choices, instead following what a related project (e.g. OpenEmbedded/Yocto/Poky) does without modifications.

8. Infrastructure Needed

N/A

9. References

N/A