Removal of Dead Code High Level Design

Added by Richard Henwood, last edited by Richard Henwood on Mar 22, 2014

Introduction

This document records the technical and practical design process for the development of the Removal of Dead Code project. Readers of this document should have also read the Removal of Dead Code Scope Statement and Removal of Dead Code Solution Architecture.

Design of Checkpatch Style Robot

The Checkpatch Style Robot will run as a completely automated software service. The robot Gerrit using the REST API and trigger `checkpatch.pl` on changes that have been updated since the last request. Once a new patch is identified in Gerrit and the Checkpatch Style Robot is triggered the robot will perform the following steps:

1. Check the patch out to a local repository.
2. Run `checkpatch.pl` against the patch and capture the out put.
3. Parse the `checkpatch.pl` output and add reviewer comments into the patch on Gerrit using the Gerrit REST API.

The code to implement the robot will pass `checkpatch.pl`.

`checkpatch.pl` version

The version of `checkpatch.pl` used by the Checkpatch style robot is a modified version of `contrib/scripts/checkpatch.pl` from the Lustre software repository. The version in the repository is not suited to machine operation and a small number of modifications to that patch must be made. These changes include:

- Add code to deprecate `cfs_wrappers`. These wrapper are currently being removed as part of ongoing independent work. `checkpatch.pl` will complain if any new uses of the wrappers are observed.

Methodology of code clean-up

The approach to removing dead code in the Lustre code base is based on a number of phases. Central to the success of this project is the development of software tools to identify candidate areas for clean-up. The Clang/LLVM tool chain is leveraged to develop the necessary software tools. These tools are created as plugins for Clang/LLVM. Once the plugins are developed the following methodology will be executed.

> **Removal priority**
> The primary task is to remove as code. By pursuing code removal first redundant work in fixing, cleaning up, re-factoring code that is subsequently deleted is avoided.

Phase 1: Applying analysis plugins against Lustre source code.

During execution, the analysis plugins for Clang/LLVM output a (long) list of undesirable code conditions including:

- unused functions

- unused variables
- unused structure members
- unused structures
- unused types
- unused macros

Phase 2: Prioritizing output from analysis

In the second phase of work, the candidate list is prioritized by an engineer according to sub-system. LLlite is the first area to be reviewed.

Phase 3: Removing un-used code

In the final phase the engineer creates a patch against Lustre source an submits it into Gerrit for review. The patch size is chosen to be small enough to reduce the chance of collisions with other patches and large enough to ensure the benefits of this work are realized as rapidly as possible.

Checking with Clang/LLVM

The Lustre file system is built from source using the Gnu Complier Collection tool chain. An alternative tool chain called Clang/LLVM has reached maturity for x86 targets. Clang/LLVM provides a powerful API to interrogate the compiler during operation. We will exploit this API to build tools to aid identifying unused code that must be removed. Changes necessary to make Lustre source checkable with Clang/LLVM will be made and landed on the Lustre source code Master branch. Clang/LLVM provides a number of code warnings out of the box. These can identify areas of concern and will be reviewed.

Design of Clang Plug-in

When gcc is called, Clang is also called with identical arguments as well as an instruction to call the plugin. Locations of declarations for variables, functions, definitions are recorded. Uses of variables, functions, definitions are recorded. These are output to a text file. Once the build is complete, the declarations are compared with usages to identify unused, dead code.

The plugin will exploit the abstract syntax tree (AST) made available in Clang. More specifically, it will be a recursive AST visitor. An introduction to the Clang abstract syntax tree is available from the page http://clang.llvm.org /docs/IntroductionToTheClangAST.html. The reference page for the AST plugin object is http://clang.llvm.org/doxygen /classclang_1_1PluginASTAction.html. Further reading on the topic of developing a static code checker using the Clang AST can be found at http://clang-analyzer.llvm.org/checker_dev_manual.html.

Availability of the Clang Plug-in

The plugin will available from the HPDD public git repo under an open source license.

Sparse Static Analyser

Sparse is a static analysis tool developed for the linux kernel. It was origionally written by Linus Torvalds. We will use an

upstream version and run it against a Lustre build. Sparse Static Analyser is executed as an add-on component of the build by calling 'make' with a suitable modifier. Early prototyping work suggests that the number of false positives from Sparse Static Analyser is small and manageable.

Design of Clean Code Metrics Robot

Clean Code metrics can be generated for each patch. The output of the style bot is used to derive a metric.

Patches to Remove Dead Code

All patches to the Lustre code base require a LU ticket to link against. Patches that remove dead code will not be treated any differently.

2 Comments

Cory Spitz

For the Sparse analysis, what's the plan to exercise all lines of code (or use cases) so that little-used paths (such as error recovery) are not labeled as unused?

John Hammond

Cory,

Sparse is a static analyzer and does not check for dead code.

Primarily I will identify dead code using static analysis. So even a function which is rarely called will be identified as used. Specifically, only when the analyzer can prove that a function is not called will it be flagged as unused.

Second I have coverage data based on our autotest suite and racer. I can combine that with the analyzer data to find code and data with no live uses. I've used this in the past to validate the automated identification process.