Removal of Dead Code Solution Architecture

Added by Richard Henwood, last edited by Richard Henwood on Feb 27, 2014

Introduction

Removal of dead code from the Lustre* file system code base is an important and urgent task for reasons stated in the Problem Statement of the Statement of Work. This document is concerned with describing the Use Cases, Requirements and System Analysis.

A important product of this project is developing tools (as plugins for Clang) that will be of value to all Lustre developers into the future.

Requirements Description

Lustre software is developed in a open and collaborative way. Code contributors have different levels of experience. Code reviewers have apply different style guidelines. Lustre software development community needs to:

- Apply style guidelines evenly to contributed patches.
- Reduce the count of unused or incorrectly lines of code.

Use Cases

A software developer needs to provide a patch for the Lustre software code base.

Checks for code style requirements are completed automatically.

The developer has deviations from the style requirements made available to them in the context of their patch.

A software developer needs to reduce the lines of unused code.

A tool is available to short list code that maybe unused under all circumstances.

The developer uses the output of the tool to create patches removing unused code.

A software developer needs to ensure header files are correctly included.

A tool is available to analyse the Lustre source code and identify redundant header inclusions.

The developer uses the output of the tool to create patches removing unused code.

Solution Proposal

Automated style-checker

A automated style-checker service will be implemented. style-checker will use the perl script (checkpatch.pl) in the Lustre software to check a patch against a set of best practice style guidelines. When an author pushes a patch into Gerrit the patch must be verified for style issues using the checkpatch.pl script. Deviations from the style must be provided in line with the patch in Gerrit. Example output is:

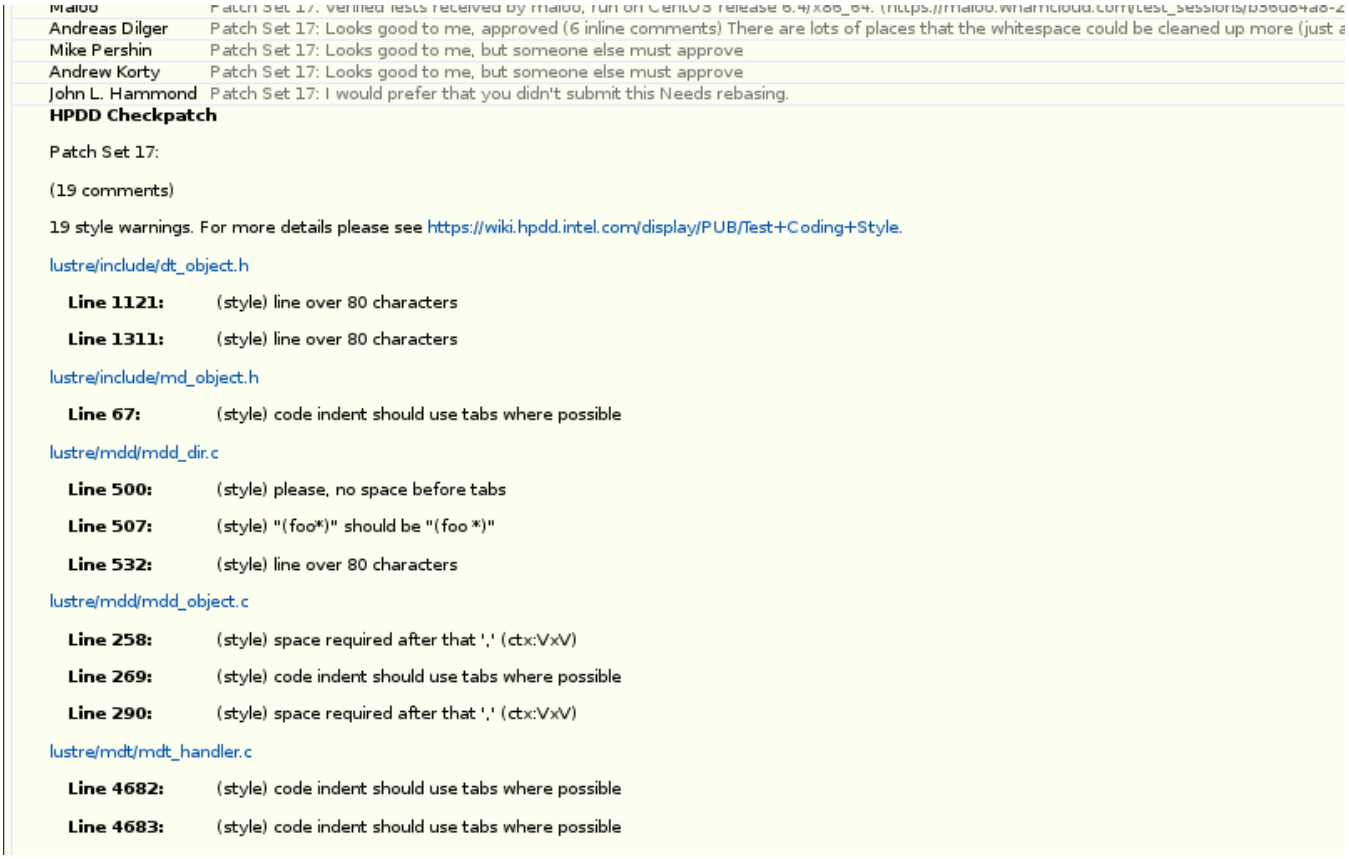

Figure 1: An example of style-checker commenting against a patch in Gerrit

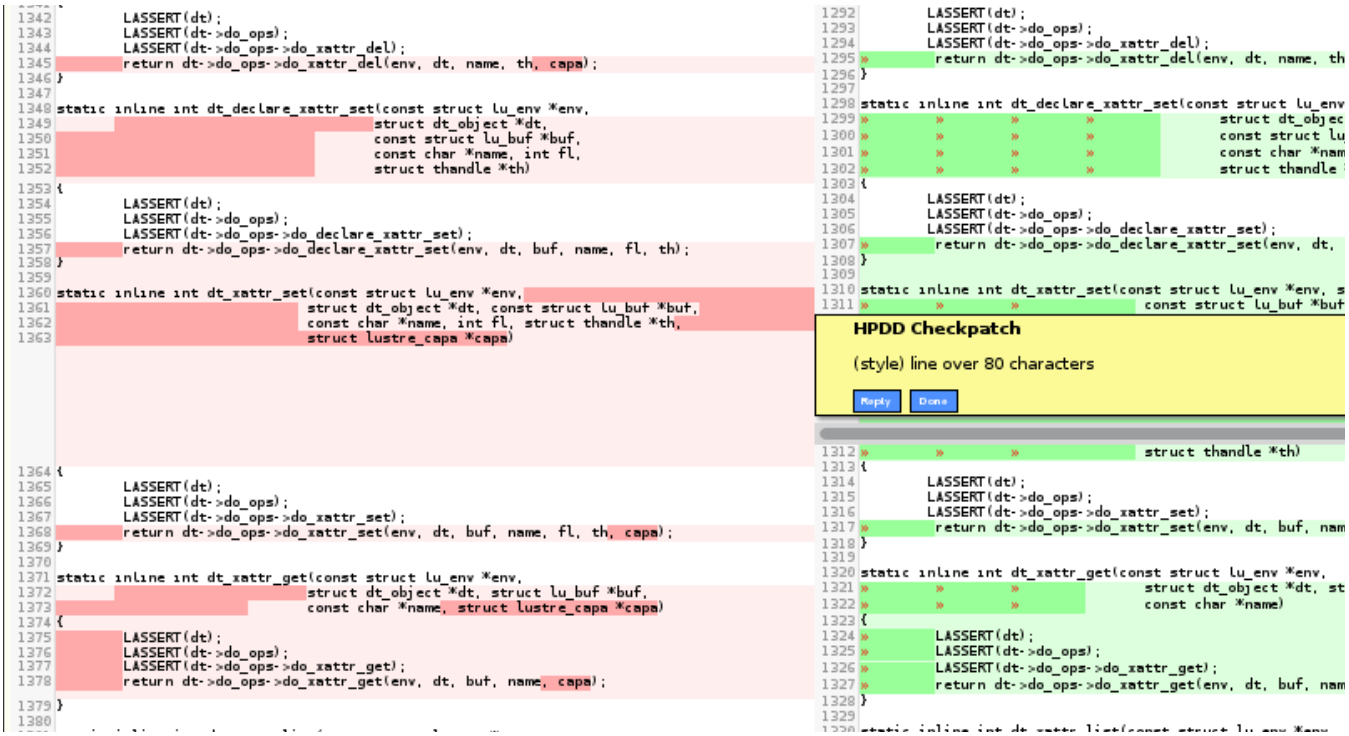

Figure 2: An example of in-line style comment from style-checker against a specific part of the patch.

Static Analysis to Identify Unused Code

We will develop a Clang plugin to identify lines of code that are never run (i.e. unused and redundant). By pairing this tool with a extensive Lustre workload, lines that are never run under the workload are revealed. This tool does not guarantee that code will never run under any circumstances. The tool provides a short-list of candidates for investigation and potential deletion by a expert developer.

Static Analysis to Identify Omitted use of 'const' in Function Prototypes

Function prototypes should have 'const' for unmodified pointers. Function prototypes that do not have 'const' for unmodified pointers need to be corrected. The CLANG and LLVM compiler tool chain will be extended with a plugin to resolve instances of incorrect const usage.

Static Analysis to Identify Redundant Header Inclusions

Files that have redundant header inclusions need to be corrected. The CLANG and LLVM compiler tool chain will be extended with a plugin to resolve instances of redundant header includes.

Sparse for Linux Specific Checks

Sparse is an open source tool designed to find coding faults in the Linux kernel source. The Sparse tool will be run against the Lustre source code. A large part of the Lustre source code is Linux modules which should follow Linux Sparse guidelines. If any issues are discovered they will be short-listed for correction.

Requirements Analysis

Automated operation for style-checker.

- The style-checker must be run immediately a new patch is pushed into Gerrit.
- The style-checker will be loosely coupled to Gerrit running on it's own machine and communicating with Gerrit over ssh.
- The style-checker will not score patches with -1 if there are issues with the style.
- If the style-checker does not execute on a newly pushed patch (because the style-checker host is down for example), the style-checker will not process any patches it missed when it returns to working order.

Static Analysis

- Static analysis tools will be run ad-hoc.
- Static analysis tools will be available to a developer to run locally.

Static Analysis tools include:

- Identifying unused code.
- Identifying incorrect use of const in function prototypes.
- Identifying incorrect header inclusions.
- Identifying Linux kernel specific coding faults with Sparse.

Acceptance Criteria

Automated check-patch operation will be demonstrated.

Ad-hoc un-used code analysis will be demonstrated.

Ad-hoc incorrect use of const in function prototypes will be demonstrated.

Ad-hoc incorrect header inclusion analysis will be demonstrated.

Ad-hoc Linux specific coding faults analysis will be demonstrated.

2 Comments

Andreas Dilger

It would be desirable for the style-checking tool to also review patches that were submitted while it is disconnected from Gerrit. Is it possible to have the list of patches to inspect based on a Gerrit query that can be run periodically instead of watching for new patches to be reviewed?

John Hammond

Yes. That was always the plan.