Autocuration: An Implementation of a Continuous Integration System Employed in the Development of AMD’s Next-generation Microprocessor Core

Wei Foong Thoo
Advanced Micro Devices, Inc.
1 AMD Place
Sunnyvale CA 94088, USA
weifoong.thoo@amd.com

David A. Burgoon
Advanced Micro Devices, Inc.
2950 E Harmony Rd #300
Fort Collins CO 80528, USA
dave.burgoon@amd.com

ABSTRACT

A mechanism for validating each source code change contribution to the version-controlled source code repository is vital in a large microprocessor design project. This mechanism is part of a software development practice known as continuous integration. This paper will describe AMD’s “Autocuration” system: the set of tools comprising the automated continuous integration system used in the development of the verification and Verilog HDL source codes for AMD’s next-generation microprocessor core.

To make efficient use of available computer resources, the system has a dynamic process to determine the appropriate subset of regression test suites that need to be run for each source commit to the trunk. It also maintains the latest overall passing version of the microprocessor core as well as the latest passing version of individual unit suites, which provides a convenient way for each sub-team responsible for a particular functional unit of the microprocessor core to choose a version of the trunk that is suitable for running nightly regression tests. The data stored in the system is also a useful source for generating metrics to track the state of the project. The architecture of the system and all its components will be presented in detail along with the supporting flows, problems in the current system, and ideas for further improvement.

General Terms

Verification, Continuous Integration System

Keywords

Verification, Nightly Regression, Release Flow

1. INTRODUCTION

In a large microprocessor design project with many team members, it is crucial to complement a robust version control system with a mechanism for validating each source code change contribution so any bad code is quickly fixed or rejected. This mechanism ensures that one can simply check out a specially designated recent copy of the source tree from the version controlled source code repository with the confidence that key builds and simulations will pass, and the copy is therefore viable for use in further development. This mechanism is part of a software development practice known as continuous integration. It is normally implemented by having a set of tools to help developers run a suite of regression tests before committing their local changes to the source code repository, and having a back-end daemon that detects the commits in order to trigger builds and simulations.

This paper will describe AMD’s “Autocuration” system: the set of tools comprising the automated continuous integration system used in the development of the verification and Verilog HDL source codes for AMD’s next-generation microprocessor core, used in conjunction with a copy-modify-merge source code control approach. The development team was organized into sub-teams based on the functional units of the core. Each unit team developed its own set of test benches and tests, which form the regression test suite for the unit. In addition to these, there was also a core-level regression test suite. If we had naively attempted to use all these test suites, every commit to the source code repository could have potentially spawned an unmanageably large set of compute-intensive jobs. To avoid this, techniques were developed to dynamically determine the appropriate sub-set of regression test suites that need to be run for each commit without sacrificing the ability to view fine grained results for each commit.

Besides providing the results to the author of each commit via e-mail notification, there is also a Web-based interface for viewing the results of the regression and other useful details for each commit such as the author, list of files modified, and a summary of the changes made relative to the previous version. All the data stored in the database of the system serves as a good source for generating various metrics to track the state of the project. The system also has a method for individually tagging the health of each unit’s regression test suite. This provides a convenient way for each unit to choose a passing version in the trunk for running its comprehensive nightly regression tests without being impacted by any commits that break the health of other units.

Having such a system in place in a project does away with the need to perform time-consuming periodic manual integration of all the units of which the core is comprised of. It provides early warning of incompatible code, and promotes regular commits, which encourages complex features to be broken up into smaller chunks [1].

The architecture of the system and all its components will be presented in detail along with the supporting flows, problems in the current system, and ideas for further improvement. The basic workflow and guidelines used by the logic design and verification engineers of this project to complement the system will also be covered. This is a system that has been proven to work successfully for a project with developers working in different parts the world with a high volume of commits, to the tune of a commit every few minutes.

2. BASIC WORK FLOW AND GUIDELINES

All engineers in the project follow a sequence of simple steps to develop source code that is ready for committing into the source code repository (Figure 1). The basic idea is that each developer begins work on a set of changes using a known-good copy of the version control repository in a private workspace, such that any build or simulation errors observed using this workspace can be safely assumed to be caused by the as-yet uncommitted changes to the source files. These must be corrected before the changes may be committed to the repository.

The first step is to run a command that creates a new workspace. By default, that command checks out from the trunk of the source code repository the latest version that passes all the regression test suites configured in the `release_gate` tool, the tool that each engineer runs to qualify their changes before committing them to the source code repository. Such a version is designated by the Autocuration system as Latest Known Good. If a workspace already exists, we run a command that incrementally updates the workspace to, by default, the Latest Known Good version.

With a workspace that is of Latest Known Good quality, local source code development can be done with the assumption that any subsequent build or simulation failures encountered are due to locally modified changes. In practice, the development engineer usually runs `release_gate` (or individual tests) one or more times to incrementally qualify the changes under development, without encountering any noise due to the incremental development of others (whose work in progress is safely sequestered in individual private workspaces). When an engineer has completed making local source code changes, it is time to run `release_gate` one final time. But before doing so, it may be important to update the files in the workspace to the Latest Known Good version of the trunk. This ensures that incompatibilities with the change-sets committed since the workspace was last created/updated are dealt with locally, and not committed until resolved. This is such an important step to do that the `release_gate` tool actually pauses to print a warning about the need to update the workspace if the current workspace version is 50 versions older than the current latest passing version. Because we print only a warning, this final update step is optional, and left to each engineer's judgment.

After the final `release_gate` completes with a passing result, the local changes can be safely committed to the source code repository. If there is at least one failure, the problem must be debugged, and any modification to the changes will have to go through the aforementioned flow before attempting to commit the changes again.

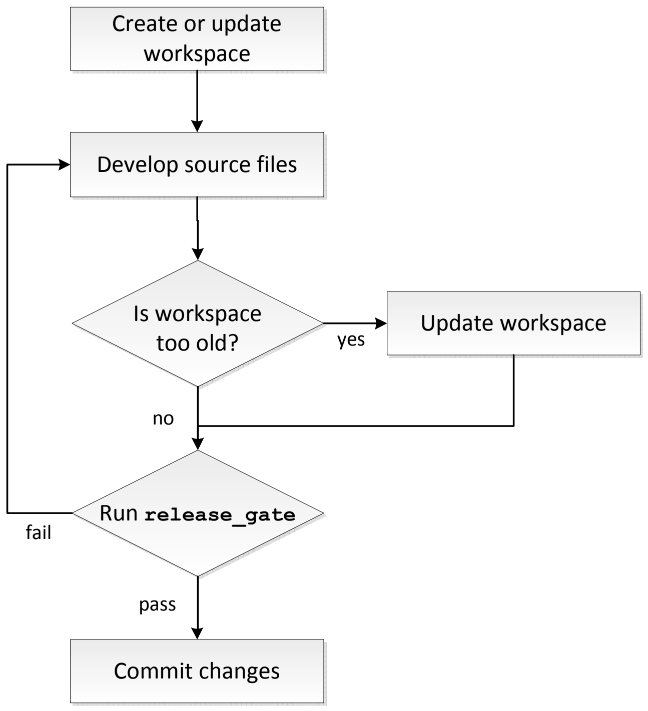


Figure 1: Basic work flow used by all engineers

Before making a commit, it is common for copy-modify-merge source code management systems to require locally modified files to be updated by merging them with the latest change-set at the head of the trunk. This update could result in merge conflicts. If the merge conflicts are minor and can be resolved easily, it is fine to proceed with the commit after resolving the conflicts. But if the conflicts are extensive, it is recommended to update the workspace to the version that introduced the conflicts and rerun `release_gate`.

It is important for the regression test suites in `release_gate` to complete within an acceptable amount of time. One reason is that `release_gate` is meant to be an interactive process in which the user eagerly waits for the results to determine that the locally made changes did not break anything. The other reason is that the same `release_gate` is also run by the Autocuration system after each commit is detected. When the `release_gate` run on the committed version passes, that version will be marked as the new Latest Known Good version. Hence, the faster it completes, the quicker a newly committed version is made available by default to other users who create or update their workspaces. This ensures that the Latest Known Good version is constantly very close to the head of the trunk, which is desirable so code change qualification is always done against a recent baseline code.

It may seem redundant for Autocuration to run `release_gate` again because the author of the commit has already done so. But the key difference is that the `release_gate` run by Autocuration includes all new commits that were made since the workspace version in which the local `release_gate` was run. It catches any incompatibility that may exist between the newly committed changes and all the other recently made commits that were introduced while the local `release_gate` was running.

There is no strict enforcement of the requirement to run `release_gate` before making a commit. Hence, an irresponsible author could potentially commit changes without running `release_gate`. However, if the commit fails in Autocuration, the offending commit will be reverted. More details about this are discussed in Section 3.6. In practice, a wise and diligent engineer may run more regression tests than what is in the standard `release_gate` suite to ensure that the changes do not introduce problematic large-scale failures in the nightly regression.

Because the sequence of updating the workspace, running `release_gate`, and committing the changes if `release_gate` passes is done so frequently, these steps are performed automatically by `release_gate` if the `-donate` switch is used. A large number of the engineers in the project choose to make their commits this way.

Another common work flow pattern is to speculatively update the workspace to a recently committed version that has not yet been marked as the Latest Known Good version. Usually this practice is done by a person who wants to commit changes made so far on a complex feature to save the current work in progress, and then wants to immediately continue working on the feature. Another use case is one in which two or more engineers working on similar areas of the code update to the newly committed version by the other engineer before beginning additional changes to avoid complicated merge conflicts.

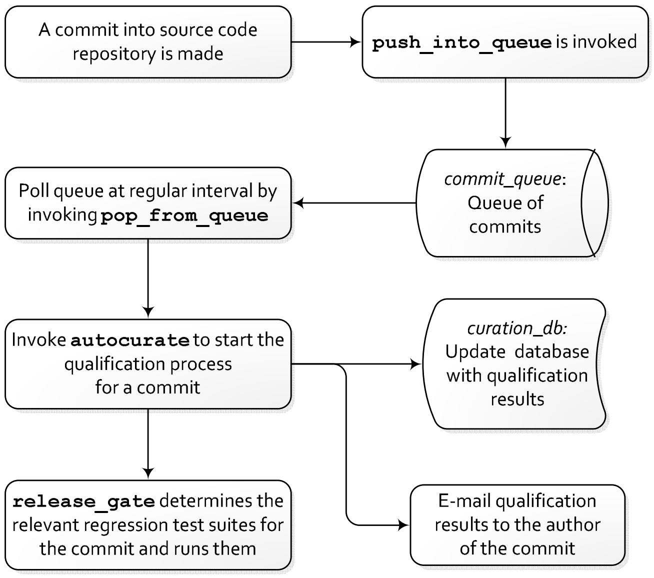
3. DETAILS OF THE SYSTEM

The Autocuration system is made up of the key components described in Table 1.

Table 1: Components of the Autocuration system

Perl scripts	*Crontab entries*
• `push_into_queue` • `pop_from_queue` • `autocurate` • `ac_cleanup` • `release_gate` • `auto_commit`	• Run `pop_from_queue` every 5 minutes • Run `ac_cleanup` every 5 hours • Run `auto_commit` every 4 hours
PHP scripts	***MySQL database tables***
• `curation_index.php` • `rm_from_queue.php`	• *commit_queue* • *curation_db*

The whole process begins with a commit into the source code repository. After that, the process continues with a series of cron jobs that execute the various Perl scripts to qualify the changes committed. Finally, the results will be updated in the MySQL database and an e-mail notification is sent to the author of the commit. The whole sequence is illustrated in Figure 2.



A commit into source code repository is made
push_into_queue is invoked
commit_queue: Queue of commits
Poll queue at regular interval by invoking pop_from_queue
Invoke autocurate to start the qualification process for a commit
curation_db: Update database with qualification results
release_gate determines the relevant regression test suites for the commit and runs them
E-mail qualification results to the author of the commit

Figure 2: Autocuration process flow diagram

3.1 Description of the Perl scripts

▪ `push_into_queue`

This script is invoked by a commit hook in the source code management system when the commit is made. It pushes the committed version into *commit_queue*, and also updates *curation_db* with the author's name, list of committed files, and the commit message. The script actually augments the raw commit message supplied by the author with additional information such as the workspace version that `release_gate` was run on and the path and geographic site where the workspace is located. These bits of information help tremendously in debugging failures and determining how the failures were introduced.

▪ `pop_from_queue`

This script is invoked every five minutes by a cron job. Its primary function is to pick the oldest entry in *commit_queue* (i.e., the oldest committed version that is not yet processed by the system), remove the picked entry from *commit_queue*, and call `autocurate` with the entry.

▪ `autocurate`

This script creates a fresh workspace of the committed version and runs `release_gate`. It updates *curation_db* with the results of the qualification, e-mails the results to the author of the commit, and marks the committed version as passing or failing. To conserve disk space, this script deletes the entire workspace if `release_gate` passed. Otherwise, it will only remove all the source files and leave the build and simulation output files for debugging purposes.

▪ `ac_cleanup`

This script is invoked every five hours by a cron job to remove any workspaces that are older than five hours. These are the workspaces that failed `release_gate`.

▪ `release_gate`

The main function of this script is to run regression test suites. There is a data file for this script (`release_gate.yml`) in YAML format that specifies all the targets in `release_gate`. For each target, the command to build the simulation model and the commands to dispatch simulations for the simulation model are specified. It also generates data files that `autocurate` uses to update *curation_db*: (a) the qualification result status file (for each unit suite) and (b) the performance data file, which contains various metrics like cycles-per-second (CPS) data, time taken to run each batch of simulations, and the time taken to get a slot on the compute farm.

▪ `auto_commit`

This script is invoked every four hours by a cron job to trigger a `release_gate` that runs all targets. Refer to Section 3.5 for more details on why this is done.

3.2 Description of the MySQL database tables

The Autocuration system has two MySQL database tables: *commit_queue* and *curation_db*. The *commit_queue* table acts as a simple queue of commits that have not been qualified yet. The *curation_db* table is used to store various information about the commit like the timestamp, author, commit version, and so on.

Here is a listing of the actual MySQL commands to create the two tables:

```
CREATE TABLE `commit_queue` (
`id` INT( 255 ) NOT NULL AUTO_INCREMENT
PRIMARY KEY,
`version` VARCHAR( 200 ),
`author` VARCHAR( 200 )
);

CREATE TABLE `curation_db` (
`id` INT( 255 ) NOT NULL AUTO_INCREMENT,
`timestamp` VARCHAR( 20 ),
`version` VARCHAR( 200 ),
`author` VARCHAR( 200 ),
`combo_status` VARCHAR( 255 ),
`status` VARCHAR( 20 ),
`files` LONGTEXT,
`msg_file` LONGTEXT,
`qual_log` LONGTEXT,
`perf_data` LONGTEXT,
`sitename` VARCHAR( 10 ),
`infra_error` VARCHAR( 10 ) DEFAULT '0',
PRIMARY KEY ( `version` ),
INDEX ( `id` )
);
```

Most of the fields in the two tables are self explanatory. However, there are a number of fields in the *curation_db* table that warrant additional description:

Table 2: Description of fields in the *curation_db* table

Field	Description
`combo_status`	This field records the `release_gate` regression suite status for each unit.
`status`	This field records the overall `release_gate` status.
`files`	This field records the list of files modified by the commit.
`msg_file`	This field records the commit message.
`qual_log`	This field records the qualification log (i.e., `release_gate` STDOUT and STDERROR output).
`perf_data`	This field records the performance data such as the CPS data, time taken to run each batch of simulations, and the time taken to get a slot on the compute farm.
`sitename`	This field records the site name from which the commit was made.
`infra_error`	This field records whether any simulation jobs suffered a failure due to an IT infrastructure problem.

3.3 Dynamic selection of the qualifications to run

The `release_gate` script runs in two modes: user mode and Autocuration mode. In user mode, the first thing it does is run a source code management command to get a list of modified files (including added and deleted files) and a list of files not versioned by the source code management system (unknown files). The next step is to compute the list of source files for each simulation model that `release_gate` recognizes. This list of source files is also known as the bill of materials (BOM), and is derived using a feature of our build system. In Autocuration mode, only the BOM computation step is done because the list of modified files can be obtained directly from the *curation_db*.

To dynamically determine the appropriate selection of qualification suites to run, `release_gate` matches the list of modified files against the BOM for each model. If any file in the list of modified files is contained in the BOM for a model, the model will be selected. There are also certain files in the workspace associated with a target in `release_gate`, such as a test list, a script to generate test cases, and the like, that wouldn't be in the BOM of the simulation model because they are not really source files. But these files definitely affect the outcome of `release_gate`, so a modification to those files must also cause the appropriate model to be selected as part of the qualification suites to run. This is achieved through a separate `qual_tuner.yml` input file in YAML format that specifies the list of targets and simulation models that a non-source file should trigger. Hence, after matching the list of modified files against the BOM for each model, `release_gate` also matches against the files specified in `qual_tuner.yml`. Finally, if there are any modified files that did not match anything, all targets in `release_gate` will be selected to be safe.

To facilitate the ability to provide distinct `release_gate` result status for each unit of the core, a group of `release_gate` targets associated with a unit is categorized as the unit suite for a unit. With this arrangement, it is necessary to spread the selection of qualification targets to all the targets of the unit suite whenever any single target of the unit suite is selected through the BOM or `qual_tuner.yml` mechanism. This ability enables the Autocuration system to mark the unit status for each commit, in addition to marking the overall status. This provides a convenient way for each unit to choose a passing version in the trunk for running its comprehensive nightly regression tests without being impacted by any commits that fail the qualification suite of other units.

3.4 Miscellaneous `release_gate` checks

Because `release_gate` has the list of unknown files and the BOM for all models, it also performs additional check for any new source files that the user has forgotten to schedule for addition to the source code management system.

Additionally, `release_gate` performs a CPS check for each simulation model to flag any CPS degradation from the CPS bar that is set for the model. It is convenient to carry out this check in `release_gate` because it already collects the CPS data for the `perf_data` field of the *curation_db* table. This check is extremely helpful to prevent slowdown in simulation speed that would otherwise be detected only later in a nightly regression if someone noticed the delay when the regression completes. Even when it's detected through the nightly regression, it would take some effort to identify the commit that caused the slowdown if the problem wasn't caught in a local `release_gate` or in Autocuration.

3.5 Result states of a qualification

Even though only a sub-set of the regression test suites in `release_gate` is run for every commit depending on the changes made, it is still possible to infer the overall result of every commit through a mechanism that inherits the result of the previous commit. This allows us to mark a qualified commit as the Latest Known Good even though only a sub-set of the targets were run, thereby optimizing compute resource usage without sacrificing the ability to have the full `release_gate` result for each commit. For this to work accurately, it is important to not have any flaw in the logic that dynamically selects the appropriate unit suites to run (i.e., BOM enumeration must be perfect).

In addition to the standard pass or fail result state, several additional states are created for the mechanism of inheriting previous results. Table 3 shows all the result states in the system.

Table 3: Autocuration result states

State	Description
P	Passed.
F	Failed.

NP	Not run, but passed. This means that the unit suite for the unit wasn't run, but it passed the last time it was run.
NF	Not run, but failed. This means that the unit suite for the unit wasn't run, but it failed the last time it was run.
U	Undetermined. If the current run completed earlier than the previous run, it will wait for some time to get the status of the previous run. If the previous run is still in progress after waiting for some time, it will time out and the current run status will just be U.
I	The `release_gate` is still in progress.

The `release_gate` run for a commit may get stuck due to a bad host on the compute farm, or it may be running very slowly because the jobs ran on a disk that is hosted on a busy file server. When this happens, a subsequent commit that is waiting to inherit the results of this stuck or slow commit will eventually time out and receive a U result state for the unit suites that require inheritance.

If the result for a commit contains a U state for a unit suite, that U state will be inherited by subsequent commits until there is a commit that triggers the unit suite to be run. Because we can mark a commit as the Latest Known Good version only if all unit suites passed (i.e., all unit suites are in either the P or NP state), the propagation of U states should not be allowed to continue for too long. This is achieved by introducing a no-op commit (a commit that introduces no material change in source code) that triggers a run of all unit suites. In our implementation, a cron job that runs the `auto_commit` script every 4 hours does this. The frequency should be adjusted based on the rate of commits in the project.

The no-op commit also helps to determine if there is a bug in the BOM-driven logic that dynamically selects the appropriate unit suites to run. A NP state for a unit suite that turns into an F state in the no-op commit indicates that one of the earlier commits should have had an F state, but it failed to happen because the unit suite incorrectly did not run.

Figure 3 shows an example of the Autocuration results Web page. It shows a listing of all commits made in chronological order, with the most recent commit at the top of the page. The ST column refers to the overall status for a commit based on the individual unit suite status. The EX, LS, ID, FP, and CU columns refer to the sub-units of AMD's microprocessor core codenamed Bulldozer [2], while the CO column refers to the regression suite status for the overall core. The PG column shows the current progress.

In this example, user_f's 3321 commit did not pass the FP unit suite and it also timed out while waiting to inherit the results from user_h's 3320 commit. Because user_f's commit modified only files that mattered to the FP and CO suites, the rest of the columns have a U state. The columns with a U state continue to propagate until a commit that runs the corresponding unit suite. The EX and LS columns turned into a non-U state in user_g's 3322 commit because that commit modified files that triggered the EX and LS unit suites. The CU column only recovered from a U state on the cron-driven no-op 3325 commit. A person who creates a new workspace would get version 3328 because the current Latest Known Good version is 3328.

Date mm-dd-yy HHMM	Commit	Author	ST	Unit Suite Status						PG
				EX	LS	ID	FP	CU	CO	
10-20-11 17:59 UTC	@3330 [qual] [diff]	user_a	I	I	I	I	I	I	I	12/33
10-20-11 17:48 UTC	@3329 [qual] [diff]	user_b	I	I	I	I	I	I	I	18/25
10-20-11 17:23 UTC	@3328 [qual] [diff]	user_c	P	NP	NP	P	NP	NP	P	DONE
10-20-11 17:12 UTC	@3327 [qual] [diff]	user_d	F	NP	NP	NF	NP	P	P	DONE
10-20-11 17:07 UTC	@3326 [qual] [diff]	user_e	F	NP	NP	F	NP	NP	P	DONE
10-20-11 17:00 UTC	@3325 [qual] [diff]	admin	P	P	P	P	P	P	P	DONE
10-20-11 16:54 UTC	@3324 [qual] [diff]	user_c	U	NP	P	P	NP	U	P	DONE
10-20-11 16:50 UTC	@3323 [qual] [diff]	user_f	U	NP	NP	U	P	U	P	DONE
10-20-11 16:48 UTC	@3322 [qual] [diff]	user_g	U	P	P	U	U	U	U	DONE
10-20-11 16:43 UTC	@3321 [qual] [diff]	user_f	U	U	U	U	F	U	P	DONE
10-20-11 16:36 UTC	@3320 [qual] [diff]	user_h	P	NP	NP	NP	NP	NP	P	DONE
10-20-11 16:25 UTC	@3319 [qual] [diff]	user_i	P	P	NP	NP	NP	NP	P	DONE

Figure 3: An illustration of the Autocuration Web Page

The “[qual]” string on the Autocuration results Web page is actually a hyperlink to a Web page that shows the progress of `release_gate`, and the “[diff]” string is a hyperlink to a Web page that shows the change-set made by the commit in the standard UNIX `diff` format. The commit number in the Commit column is a hyperlink to a Web page that shows the commit change log message and the list of files modified.

3.6 Team of curators
Occasionally, somebody will make a bad commit that fails one of the unit suites’ regressions, either due to lack of appropriate local qualification or due to incompatibility with one of the earlier committed change-sets. To ensure that the head of the trunk is always passing, the bad commit must be reverted or fixed. Sometimes, the author of a commit will attempt to fix the problem he or she introduced, which is what user_f’s 3323 commit did in our example in Figure 3.

However, to ensure that the problem is fixed in a timely fashion, a team of curators, with one representative from each unit, is formed to share the responsibility of periodically monitoring the health of the trunk. If a bad commit is detected, the curator will either try to fix it if the fix is very simple and obvious, or just revert the commit. In our example, user_c’s 3328 commit reverts user_e’s bad 3326 commit that broke ID’s unit suite.

3.7 Disk and server management
Because the Autocuration system continuously performs a very high volume of disk-intensive activity, it can overload a file server if everything is done on a single partition on the same server. To mitigate this problem, the Autocuration disks are broken into multiple partitions that are hosted on different file servers. The disk for running the `release_gate` on each commit is then picked in a round-robin fashion.

Two dedicated compute servers were used to run the cron job that executes `pop_from_queue` every 5 minutes, staggered alternately on each server. This was done because a single server can get overloaded with the number of parallel `release_gate` processes that are running at any point in time.

3.8 Metrics generation
The data stored in *curation_db* is a good source for generating various charts about state of the project. Examples are charts that show the average commits per day, average time between commits, the time taken for the various builds to complete, the time taken for the simulations to complete, and the time spent waiting to get a machine slot in the compute farm. These metrics assist in identifying problems in the design infrastructure that impedes efficiency.

4. PROBLEMS AND IMPROVEMENT IDEAS

4.1 Difficulty in identifying the bad commit
One problem with the system is the difficulty in figuring out which commit introduced a failure if another bad commit was made in the shadow of an earlier bad commit. For example, if commit 1103 causes the compilation of simulation models to fail, the next several commits won’t be able to run any simulations. If a subsequent commit 1106 causes simulations to fail, we won’t be able to easily tell which commit started the simulation failures even if the original bad commit 1103 was reverted. In practice, this is not a major problem because the bad commit is usually manually reverted by the team of curators in a timely fashion. But to fully resolve this problem in an automated fashion, the idea described in Section 4.5 can be implemented.

4.2 New releases can be blocked by a bad commit
If a bad commit was left in the trunk for too long, it causes the Latest Known Good version to become older and older as time passes. Because the creation and update of a workspace uses this version by default, engineers in the project are essentially blocked from picking up new changes released by their peers. Even though it is possible to manually update to a version that has failures to bring in changes released by others, it is still not easy to qualify locally made changes properly with the presence of existing failures. This is not a big issue because a bad commit wouldn’t stay in the trunk for too long with a team of curators keeping a close watch. This problem can also be resolved by the idea described in Section 4.5.

4.3 Further reduction of resource usage
A possible enhancement to the system to further reduce the usage of computer resources is by running `release_gate` once for several commits. The list of modified files for the set of commits is combined for the purpose of determining which unit suite regressions to run. The `release_gate` result can then be applied to all commits in the set. If a failure was detected, `release_gate` should be run for each commit individually to identify the commit that introduced the failure.

4.4 Getting rid of U states
Even though it is not a real problem, the U states (light blue in Figure 3) scattered around the Autocuration results Web page makes it a little confusing to a person viewing the Web page as compared to a Web page with just a passing or failing state with a green or red background color. One way to improve this is by having a process that periodically walks through the result states for all commits that completed running `release_gate` to convert all the U states to either NP or NF states.

4.5 Staging commits on a branch
The Autocuration system only verifies the quality of the commits after they have been committed to the trunk of the source code repository. Once a bad commit has been made, rolling it back is purely a manual process. Until it is rolled back, the bad changes tend to propagate into user workspaces as a consequence of the conditionally mandatory merge at commit time that is a natural part of copy-modify-merge version control systems. As an improvement, the system can be enhanced with a flow that sequesters each change on a branch, rather than staging potentially bad changes on the trunk. The system still verifies all change-sets through `release_gate`, but it would only release passing change-sets into the trunk through an automated branch-to-trunk merge process. All commits that fail will be abandoned (i.e., left on their branch without merging to the trunk) after sending e-mail notification to the offending authors.

5. SUMMARY
The Autocuration system is an invaluable tool relied on by all logic design and verification engineers of the project in their daily work. It makes efficient use of resources to provide fine-grained details about the health of each commit, and also provides individual latest passing tag for each unit, which facilitates the nightly regression flow. The system has been proven to work for a large microprocessor design project with engineers spread across many geographical sites.

6. REFERENCES

[1] "Continuous integration." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 9 November 2011. Web. <http://en.wikipedia.org/wiki/Continuous_integration>

[2] M. Butler, “Bulldozer – a new approach to multi-thread compute performance,” *the IEEE 22nd HotChips conference – a symposium on high performance chips*, Session 7.2, August 22 – 24, 2010.